

Artificial Intelligence: A Modern Approach

Chapter 3

Solving Problems By Searching

Outline

- ◆ Problem-solving agents
- ◆ Example Problems
- ◆ Problem formulation
- ◆ Search Algorithms
- ◆ Uninformed Search Strategies
- ◆ Informed (Heuristic) Search Strategies
- ◆ Heuristic Functions

Problem-solving-agents

A **problem-solving agent** is a goal-based agent that:

- Formulates a goal
- Formulates a problem
- Searches for a sequence of actions
- Executes the solution

Formal Components

A problem is defined by:

- Initial state
- Actions
- Transition model
- Goal test
- Path cost function

The agent assumes:

- Fully observable environment
- Deterministic actions
- Static environment
- Discrete states

It performs **state-space search** (e.g., BFS, DFS, A*, etc.).

It searches through a graph of states to find a path from the initial state to a goal state.

Example: Romania

On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest

Formulate goal:

be in Bucharest

Formulate problem:

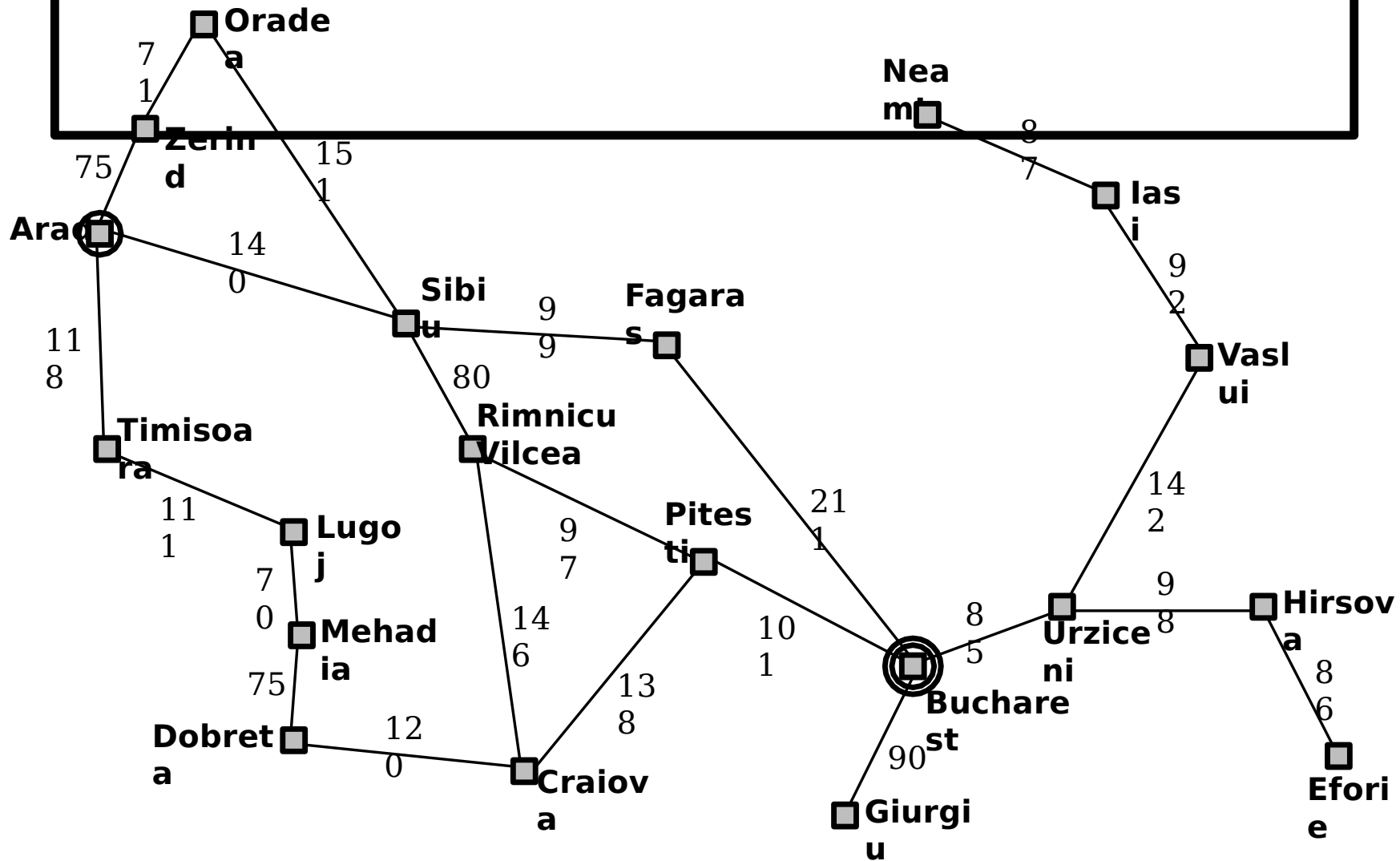
states: various cities

actions: drive between cities

Find solution:

sequence of cities, e.g.,
Arad, Sibiu, Fagaras,
Bucharest

Example: Romania



Problem-solving-agents

A **problem-solving agent** is a goal-based agent that:

- Formulates a goal: **Reaching Bucharest**
- Formulates a problem: **Devises the states and actions needed to reach the goal (model). Example: actions: traveling to an adjacent city until reaching the destination**
- Searches for a sequence of actions: **Before taking any action, the agent will simulate sequences of actions in its model.**
- Executes the solution: **The agent moves and executes the action in the solution, one at a time.**

The agent assumes:

- Fully observable environment
- Deterministic actions
- Static environment
- Discrete states

The solution to any problem is a fixed sequence of action

Example: Arad->Sibiu->Fagaras->Bucharest

If the model is correct, it can ignore perception while executing its action, this is called an **open-loop system**

If the environment is not deterministic, and an error can happen, the

Formal Components

state space: The set of possible states that the environment can be.

A problem is defined by:

Initial state: Arad, **goal state** defined by rule, example: reach any airport with a direct flight to Lakeland

Actions: available to the agent in a given state, $ACTION(s)$ return a set of actions that can be executed.

Example: $ACTIONS(Arad) = \{ToSibiu, ToTimisoara, ToZeromd\}$

Transition model: $RESULT(s, a)$ returns the **state** that results from doing action **a**, (s, a, s') for example"

$RESULT(Arad, ToZerind) = Zerind$

Action cost function: $ACTION-COST(s, a, s')$

If the problem is:

Initial state: Arad

Goal: Reach Bucharest

Then the goal test is simply

$GOAL-TEST(s) = (s == Bucharest)$ T/F

Example: If the goal is to reach any airport that has a direct flight to Lakeland

$GOAL-TEST(s) = \text{True}$ if s has a direct flight to Lakeland

Here, the goal is defined **intentionally (by a property)**, not by naming a single state

Problem types

Deterministic, fully observable $\Rightarrow$ single-state problem

Agent knows exactly which state it will be in; solution is a sequence

Non-observable $\Rightarrow$ conformant problem

Agent may have no idea where it is; solution (if any) is a sequence

Nondeterministic and/or partially observable $\Rightarrow$ contingency problem

percepts provide new information about current state

solution is a contingent plan or a policy

often interleave search, execution

Unknown state space $\Rightarrow$ exploration problem (“online”)

Problem-solving agents

Restricted form of general agent:

function Simple-Problem-Solving-Agent(*percept*) returns an action

static: *seq*, an action sequence, initially empty
state, some description of the current world state
goal, a goal, initially null
problem, a problem formulation

state ← Update-State(*state*, *percept*)

if *seq* is empty then

goal ← Formulate-Goal(*state*)

problem ← Formulate-

Problem(*state*, *goal*) *seq* ←

Search(*problem*)

action ← Recommendation(*seq*,

state) *seq* ← Remainder(*seq*,

state)

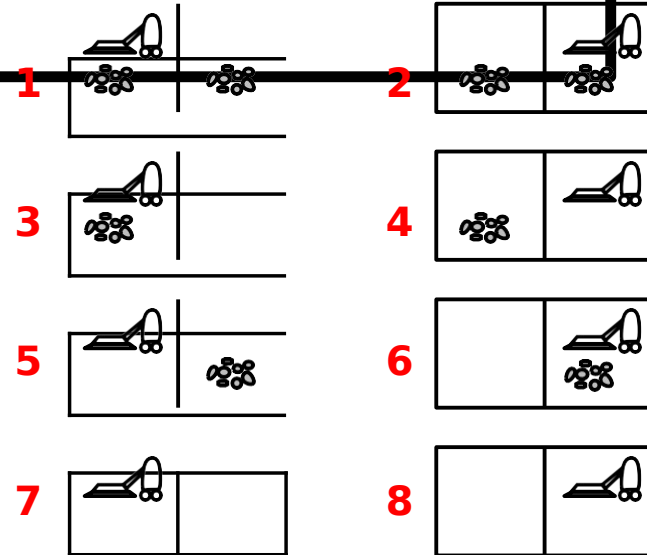
return *action*

Note: this is **offline** problem solving; solution executed
“eyes closed.” **Online** problem solving involves acting
without complete knowledge.

Example: vacuum world

Single-state, start in #5.

Solution??



Example: vacuum world

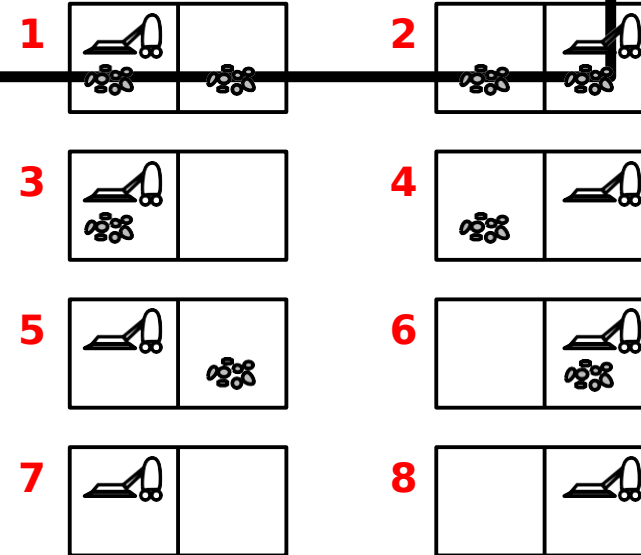
Single-state, start in #5. Solution??

[*Right, Suck*]

Conformant, start in {1, 2, 3, 4, 5, 6, 7, 8}

e.g., *Right* goes to {2, 4, 6, 8}.

Solution??



Example: vacuum world

Single-state, start in #5. Solution??

[*Right, Suck*]

Conformant, start in {1, 2, 3, 4, 5, 6, 7, 8}

e.g., *Right* goes to {2, 4, 6, 8}.

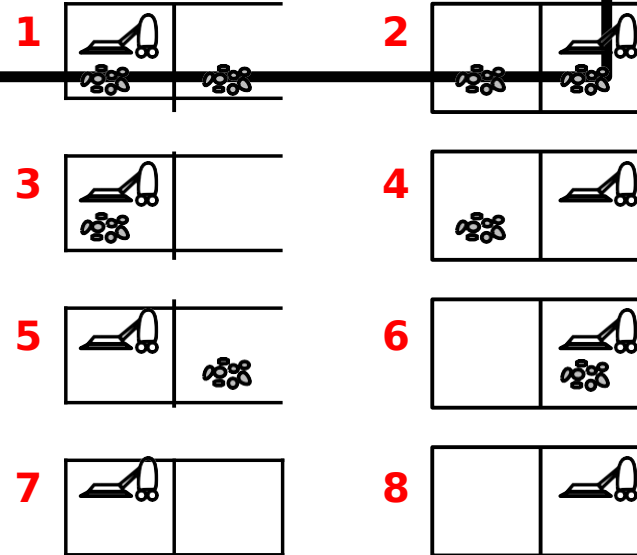
Solution??

[*Right, Suck, Left, Suck*]

Contingency, start in #5

Murphy's Law: *Suck* can dirty a clean carpet
Local sensing: dirt, location only.

Solution??



Example: vacuum world

Single-state, start in #5. Solution??

[*Right, Suck*]

Conformant, start in {1, 2, 3, 4, 5, 6, 7, 8}

e.g., *Right* goes to {2, 4, 6, 8}.

Solution??

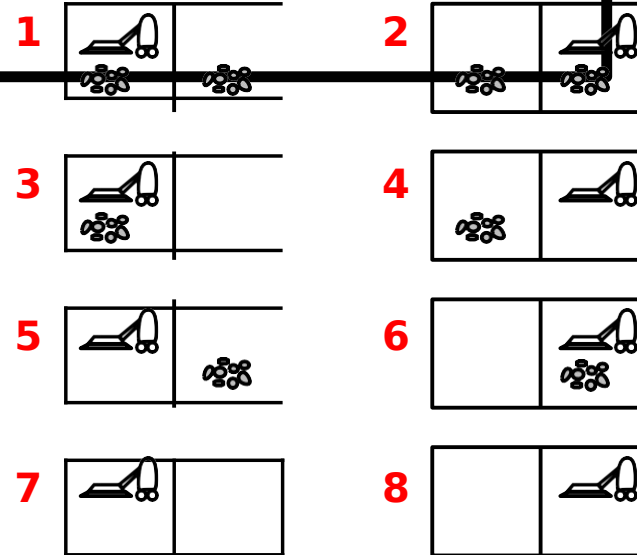
[*Right, Suck, Left, Suck*]

Contingency, start in #5

Murphy's Law: *Suck* can dirty a clean carpet
Local sensing: dirt, location only.

Solution??

[*Right, if dirt then Suck*]



Single-state problem formulation

A **problem** is defined by four items:

initial state e.g., “at Arad”

successor function $S(x)$ = set of action-state pairs

e.g., $S(Arad) = \{(Arad \rightarrow Zerind, Zerind), \dots\}$

goal test, can be

explicit, e.g., $x = \text{“at Bucharest”}$ **implicit**, e.g., $N oDirt(x)$

path cost (additive)

e.g., sum of distances,
number of actions executed,
etc.

$c(x, a, y)$ is the **step cost**,
assumed to be ≥ 0

A **solution** is a sequence of
actions

Selecting a state space

Real world is absurdly complex

⇒ state space must be **abstracted** for problem solving

(Abstract) state = set of real states

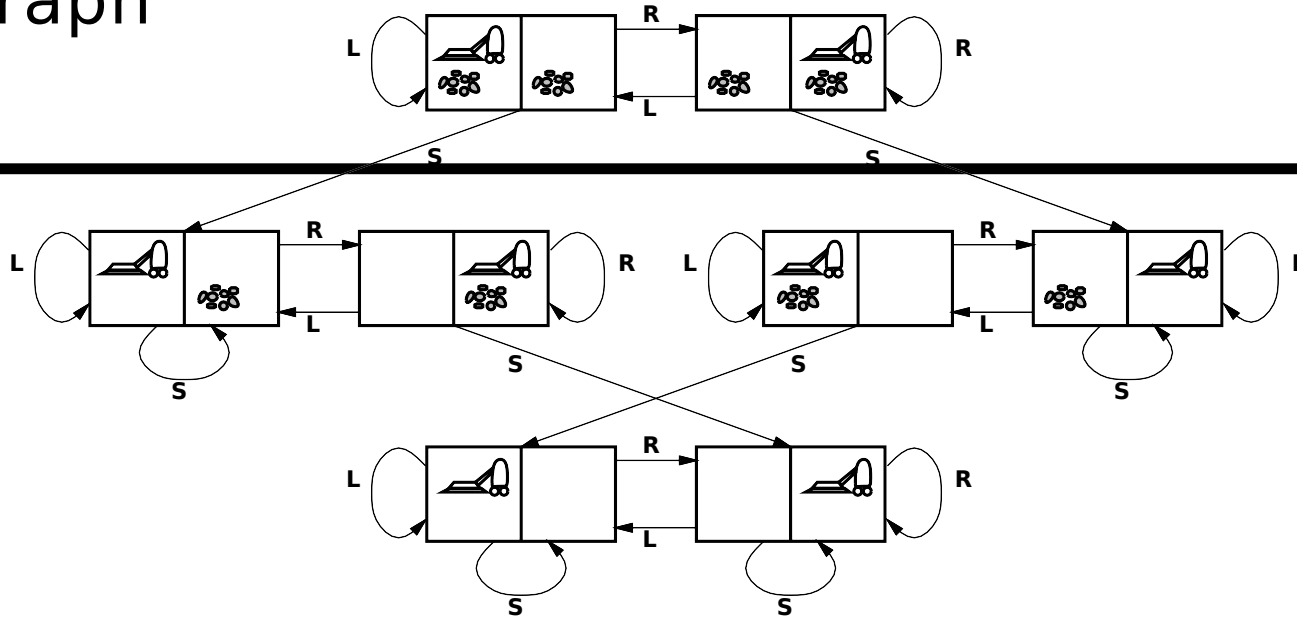
(Abstract) action = complex combination of real actions e.g., “Arad → Zerind” represents a complex set

of possible routes, detours, rest stops, etc. For guaranteed realizability, **any** real state “in Arad” must get to **some** real state “in Zerind”

(Abstract) solution = set of real paths that are solutions in the real world

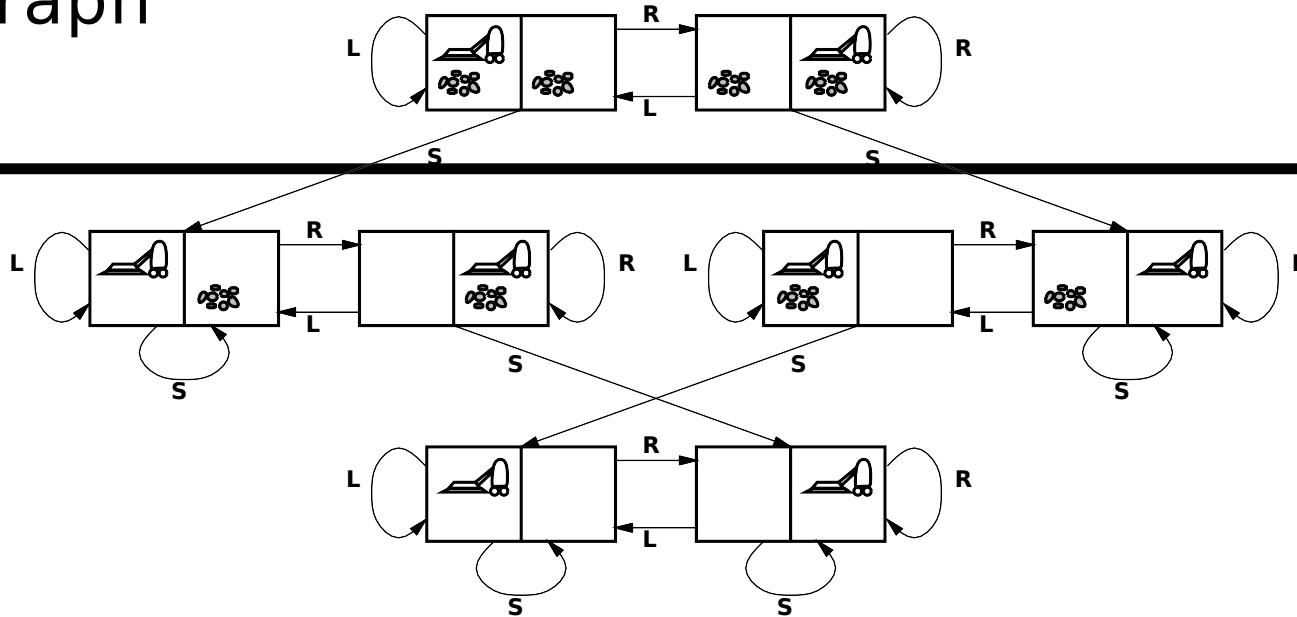
Each abstract action should be “easier” than the original problem!

Example: vacuum world state space graph



states??
actions??
goal
test??
path
cost??

Example: vacuum world state space graph



states??: integer dirt and robot locations (ignore dirt amounts etc.) actions??

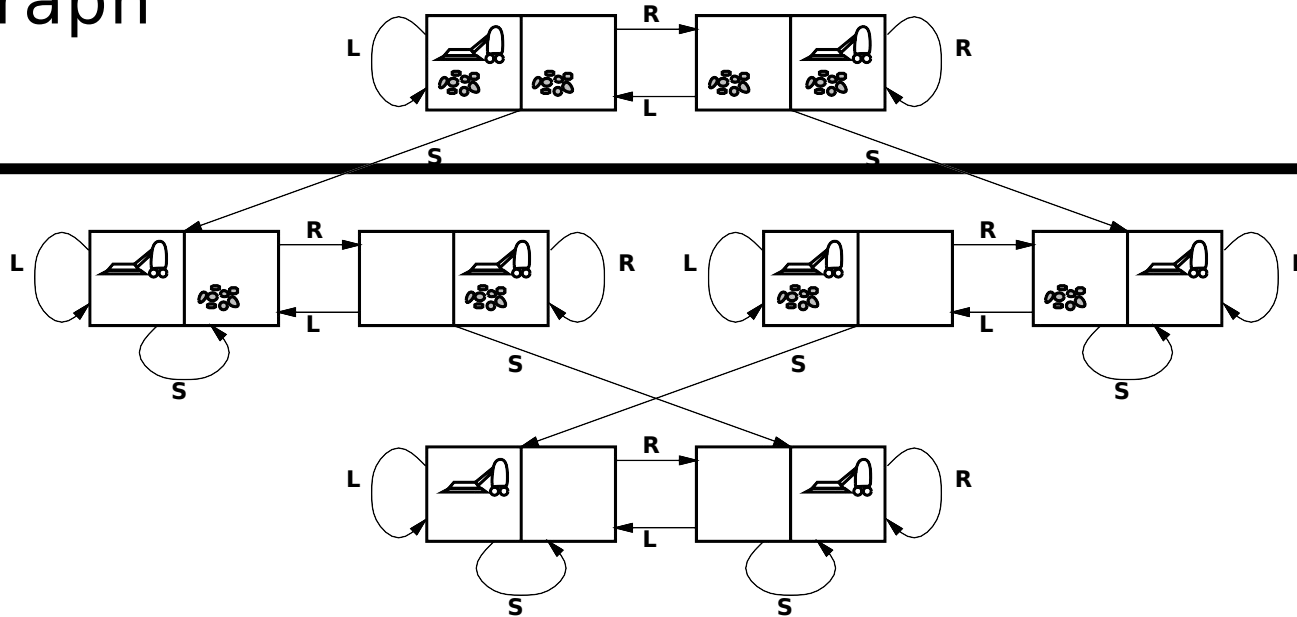
goal

test??

path

cost??

Example: vacuum world state space graph



states??: integer dirt and robot locations (ignore dirt amounts etc.) actions??: *Left, Right, Suck, NoOp*

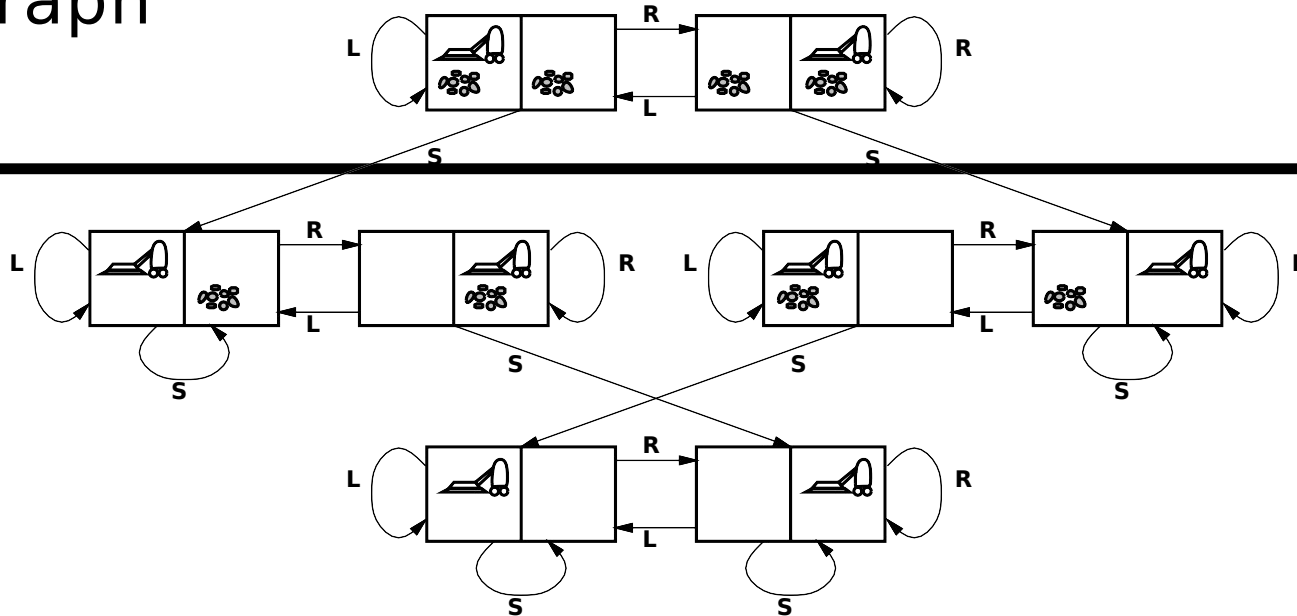
goal

test??

path

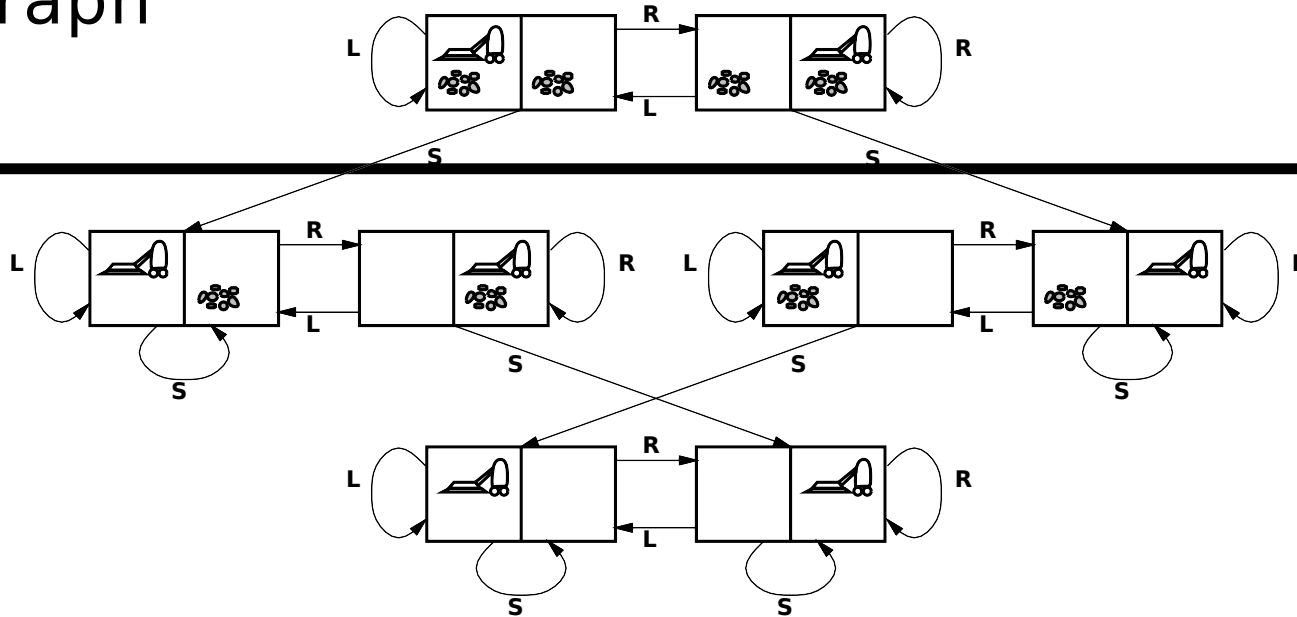
cost??

Example: vacuum world state space graph



states??: integer dirt and robot locations (ignore dirt amounts etc.) actions??: *Left, Right, Suck, NoOp*
goal test??: no dirt path cost??

Example: vacuum world state space graph



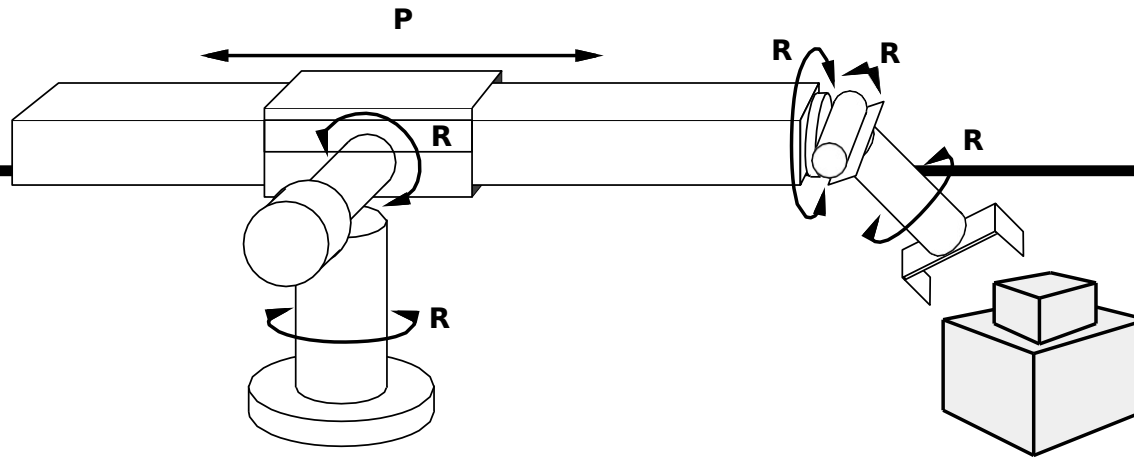
states??: integer dirt and robot locations (ignore dirt

amounts etc.) actions??: *Left, Right, Suck, NoOp*

goal test??: no dirt

path cost??: 1 per action (0 for *NoOp*)

Example: robotic assembly



states??: real-valued coordinates of robot
joint angles parts of the object to be
assembled

actions??: continuous motions of robot joints

goal test??: complete assembly **with no**
robot included!

path cost??: time to execute

Tree search algorithms

Basic idea:

offline, simulated exploration of state

space

by generating successors of already-

explored states (a.k.a. **expanding**

function **TreeSearch**(*problem*, *strategy*) **returns** a
solution, or failure initialize the search tree using the initial
state of *problem*

loop do

if there are no candidates for expansion **then**

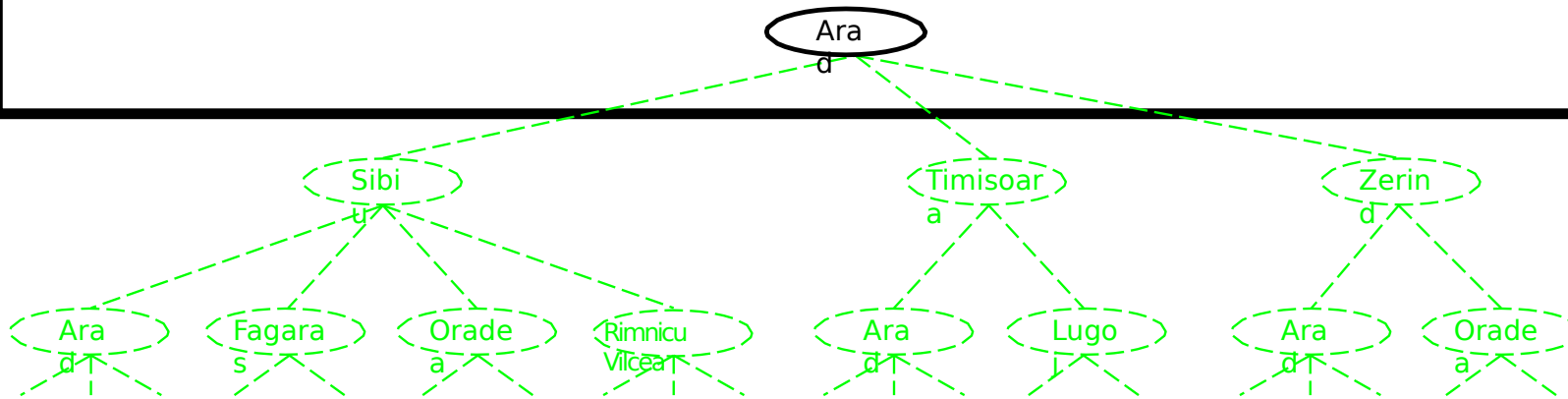
return failure choose a leaf node for expansion
according to *strategy*

if the node contains a goal state **then return** the
corresponding solution

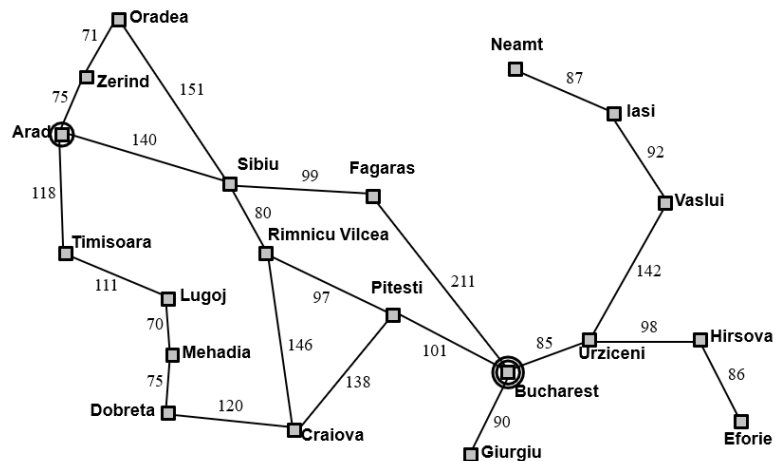
else expand the node and add the resulting nodes
to the search tree

end

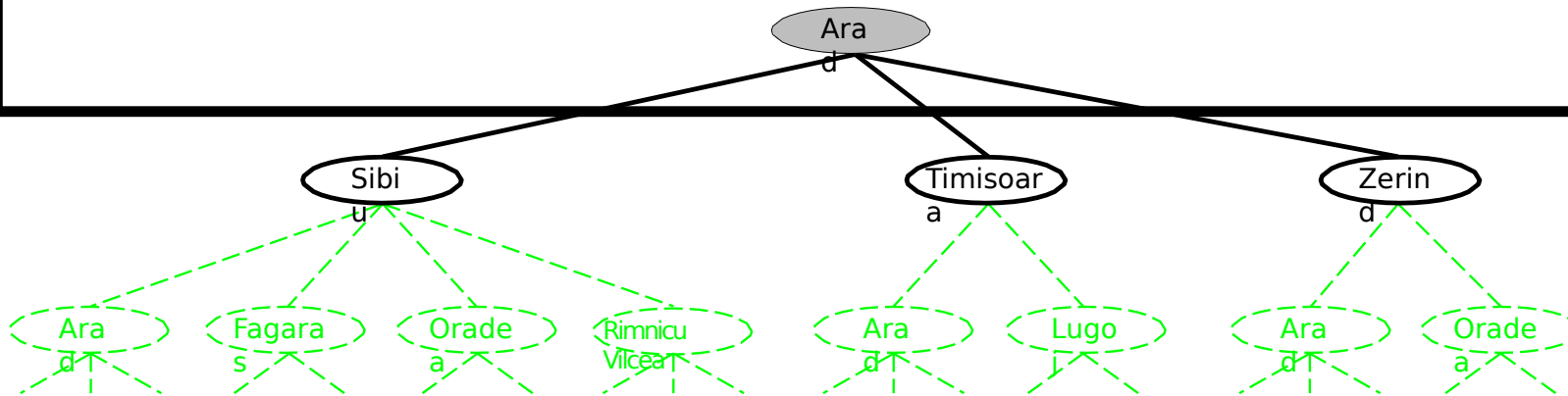
Tree search example



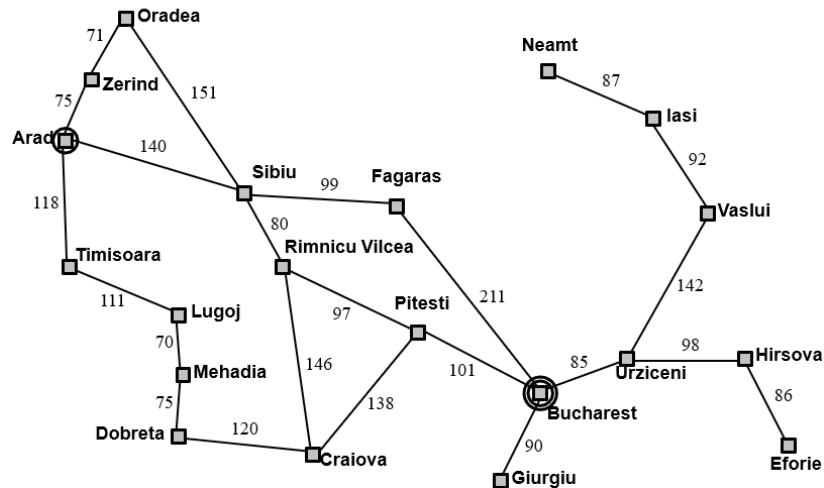
Example: Romania



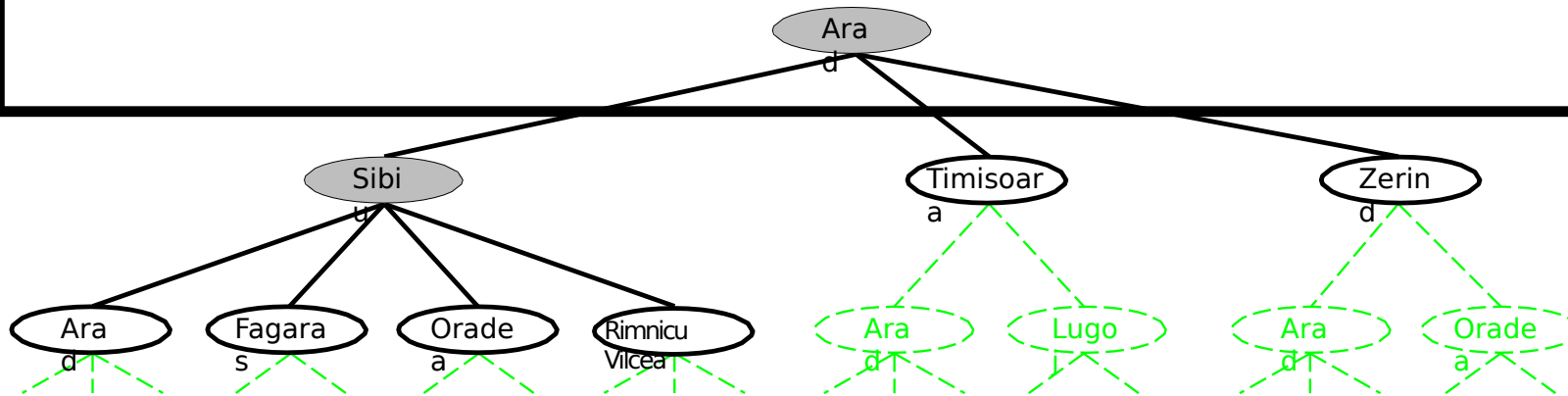
Tree search example



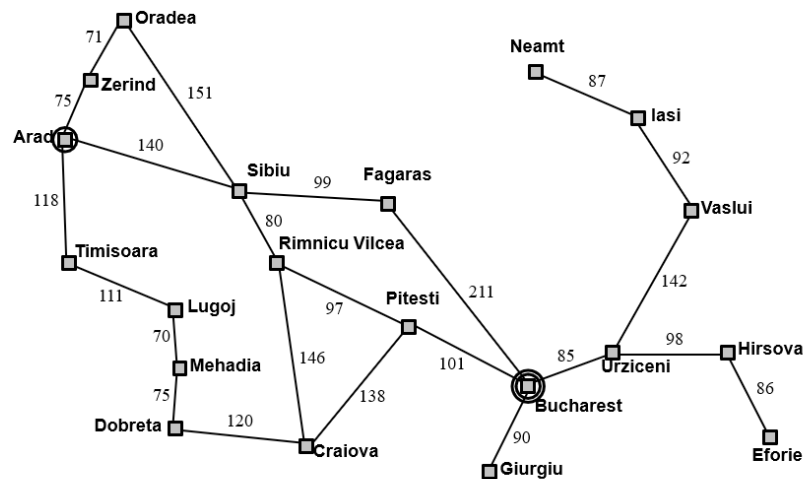
Example: Romania



Tree search example



Example: Romania



Search strategies

A strategy is defined by picking the **order of node expansion**

Strategies are evaluated along the following dimensions:

- completeness**—does it always find a solution if one exists?
- time complexity**—number of nodes generated/expanded
- space complexity**—maximum number of nodes in memory
- optimality**—does it always find a least-cost solution?

Time and space complexity are measured in terms of

- b —maximum branching factor of the search tree
- d —depth of the least-cost solution

m —maximum depth of the state space (may be ∞)

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

Breadth-first

search Uniform-

cost search

Depth-first search

Depth-limited

search

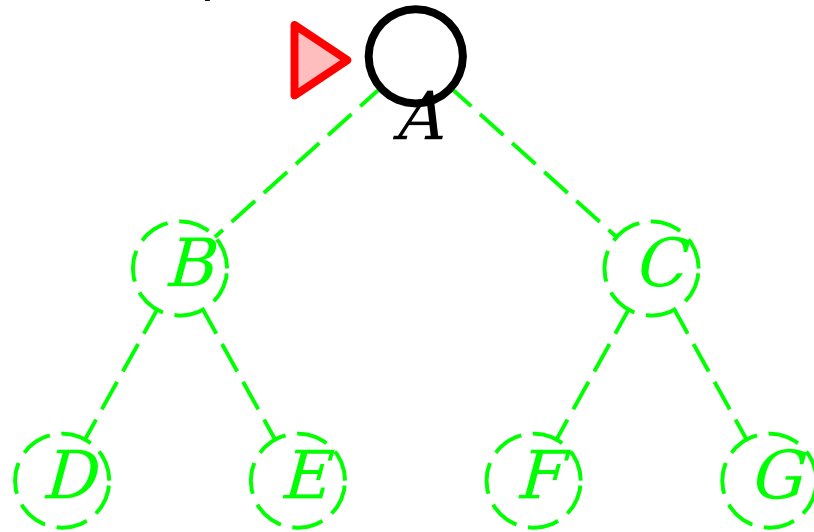
Iterative
deepening
search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

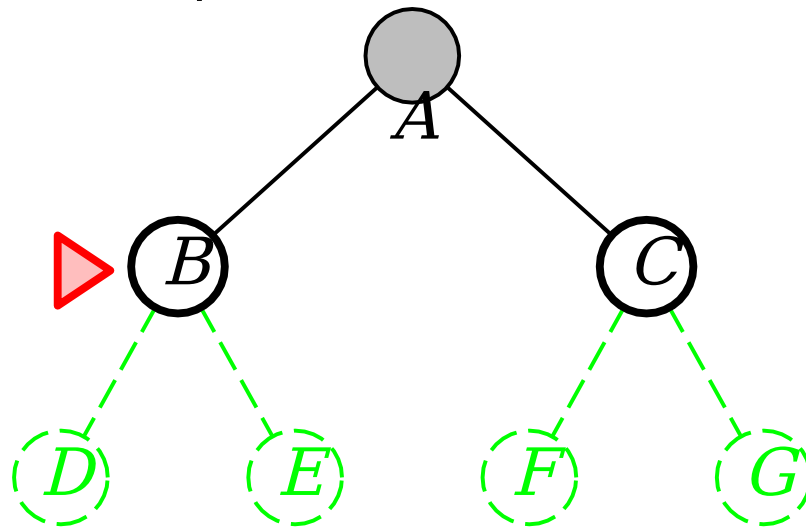


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

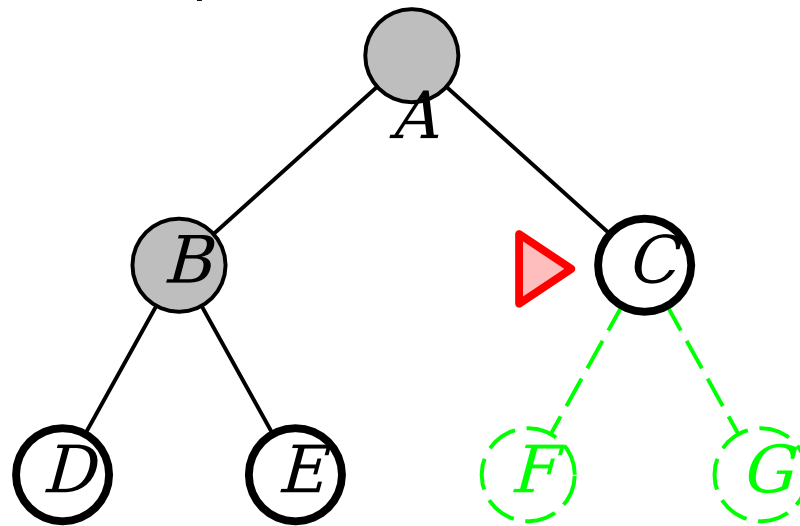


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

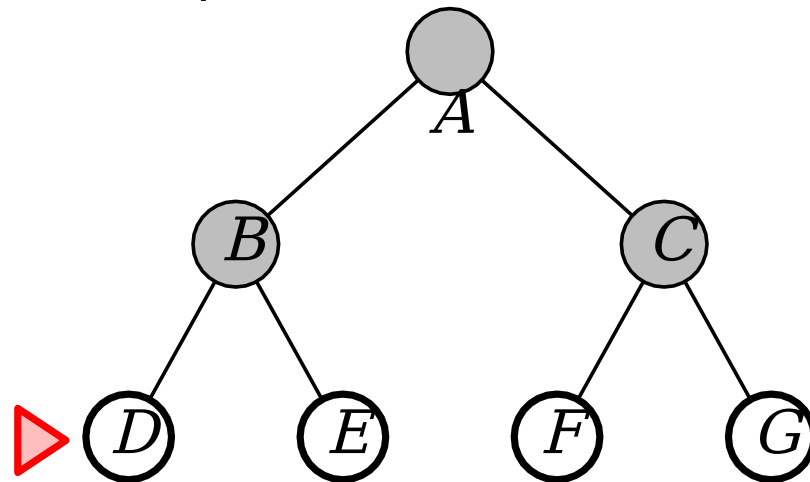


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end



BFS

The Code

BFS (graph G , start vertex s)

[all nodes initially unexplored]

-- mark s as explored

-- let Q = queue data structure (FIFO), initialized with s

-- while $Q \neq \phi$:

-- remove the first node of Q , call it v

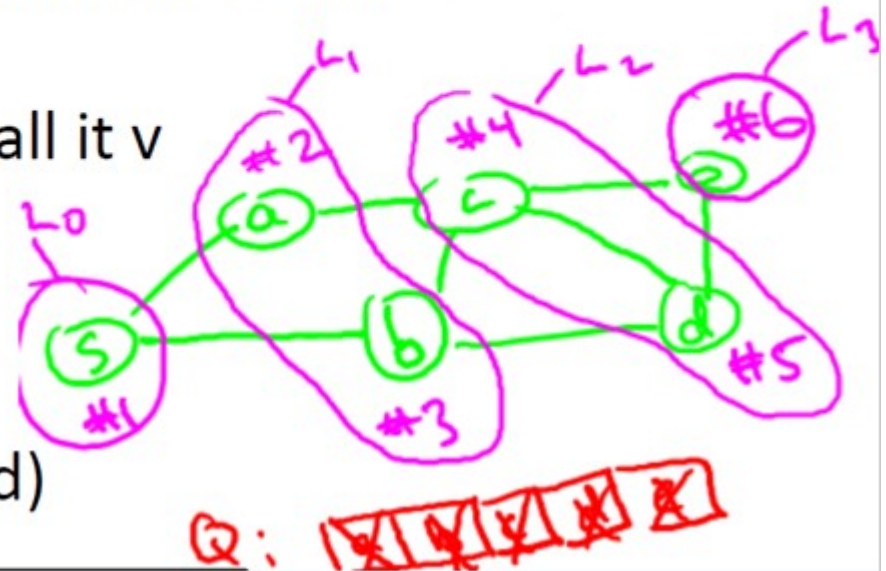
-- for each edge(v, w) :

-- if w unexplored

-- mark w as explored

-- add w to Q (at the end)

$O(1)$
time



BFS

Figure 4.3 Breadth-first search.

procedure `bfs`(G, s)

Input: Graph $G = (V, E)$, directed or undirected; vertex $s \in V$

Output: For all vertices u reachable from s , $\text{dist}(u)$ is set to the distance from s to u .

for all $u \in V$:

$\text{dist}(u) = \infty$

$\text{dist}(s) = 0$

$Q = [s]$ (queue containing just s)

while Q is not empty:

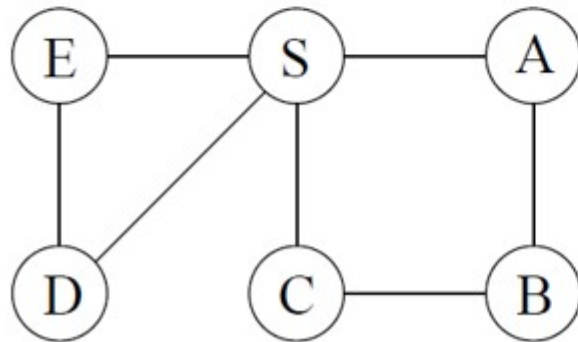
$u = \text{eject}(Q)$

 for all edges $(u, v) \in E$:

 if $\text{dist}(v) = \infty$:

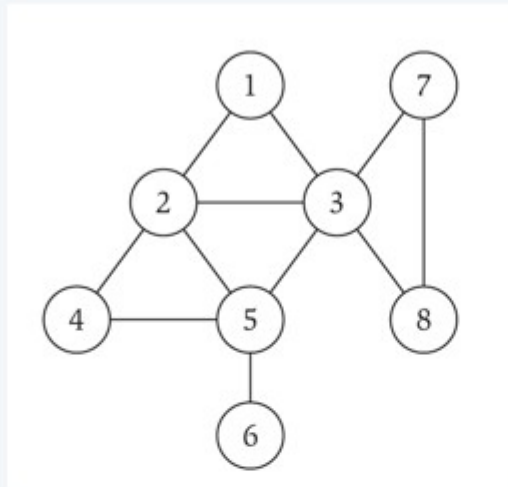
$\text{inject}(Q, v)$

$\text{dist}(v) = \text{dist}(u) + 1$



Order of visitation	Queue contents after processing node
	[S]
S	[A C D E]
A	[C D E B]
C	[D E B]
D	[E B]
E	[B]
B	[]

Property. Let T be a BFS tree of $G = (V, E)$, and let (x, y) be an edge of G . Then, the levels of x and y differ by at most 1.



procedure **bfs**(G, s)

Input: Graph $G = (V, E)$, directed or undirected; vertex $s \in V$
 Output: For all vertices u reachable from s , $\text{dist}(u)$ is set to the distance from s to u .

for all $u \in V$:

$\text{dist}(u) = \infty$

$\text{dist}(s) = 0$

$Q = [s]$ (queue containing just s)

while Q is not empty:

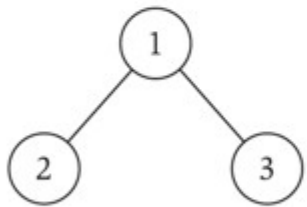
$u = \text{eject}(Q)$

 for all edges $(u, v) \in E$:

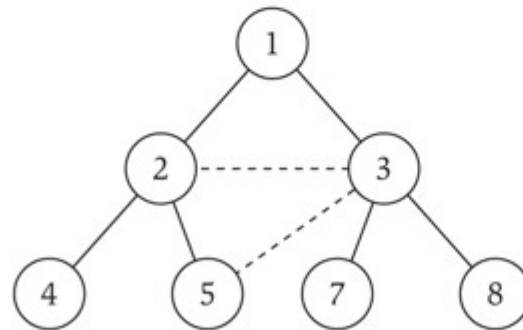
 if $\text{dist}(v) = \infty$:

$\text{inject}(Q, v)$

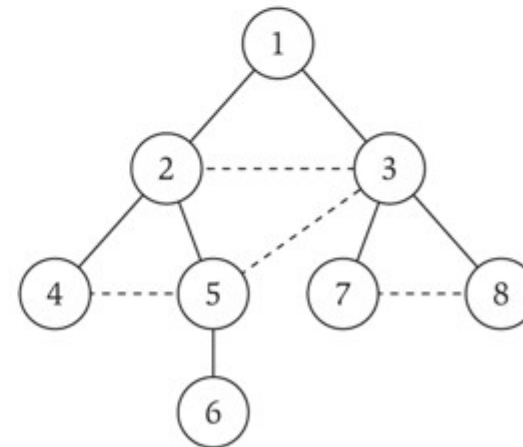
$\text{dist}(v) = \text{dist}(u) + 1$



(a)



(b)



(c)

L_0

L_1

L_2

L_3

DFS (Using a Stack)

```
procedure DFS(G, s)

stack = [s]
visited = ∅

while stack is not empty do
    u = pop(stack)

    if u not in visited then
        mark u as visited
        process(u)

    for each v adjacent to u do
        push(stack, v)
```

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$,
i.e., exp. in d

Space??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$,
i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in
memory) Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$,
i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? Yes (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at
100MB/sec so 24hrs = 8640GB.

Uniform-cost search

Expand least-cost unexpanded node

Implementation:

fringe = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs

all equal Complete?? Yes, if step cost $\geq$

c

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{fC^*/d})$

where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{fC^*/d})$

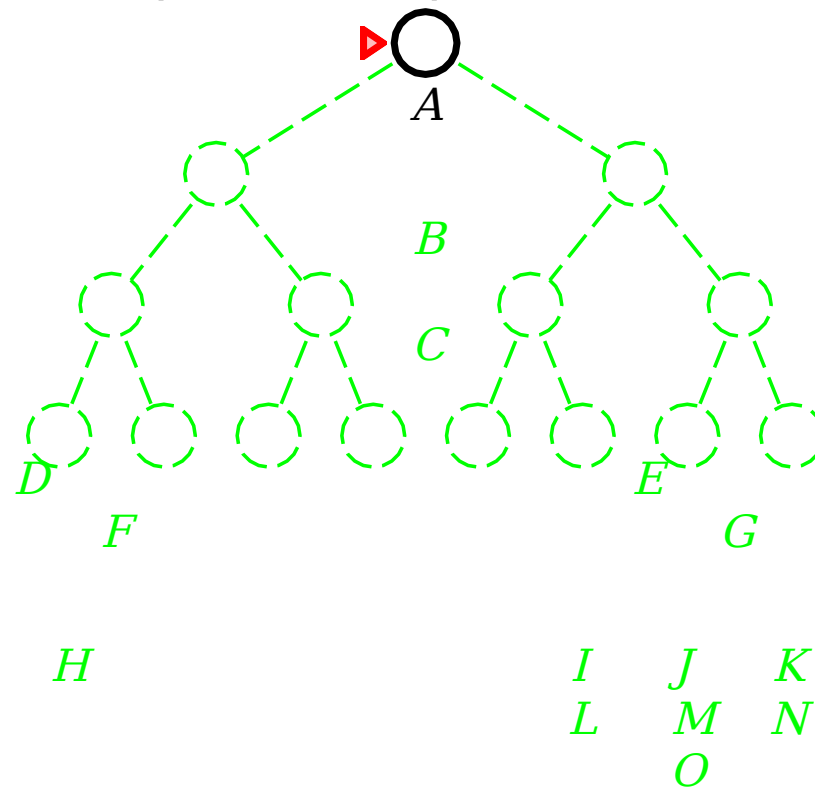
Optimal?? Yes—nodes expanded in
sing order of $g(n)$

Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

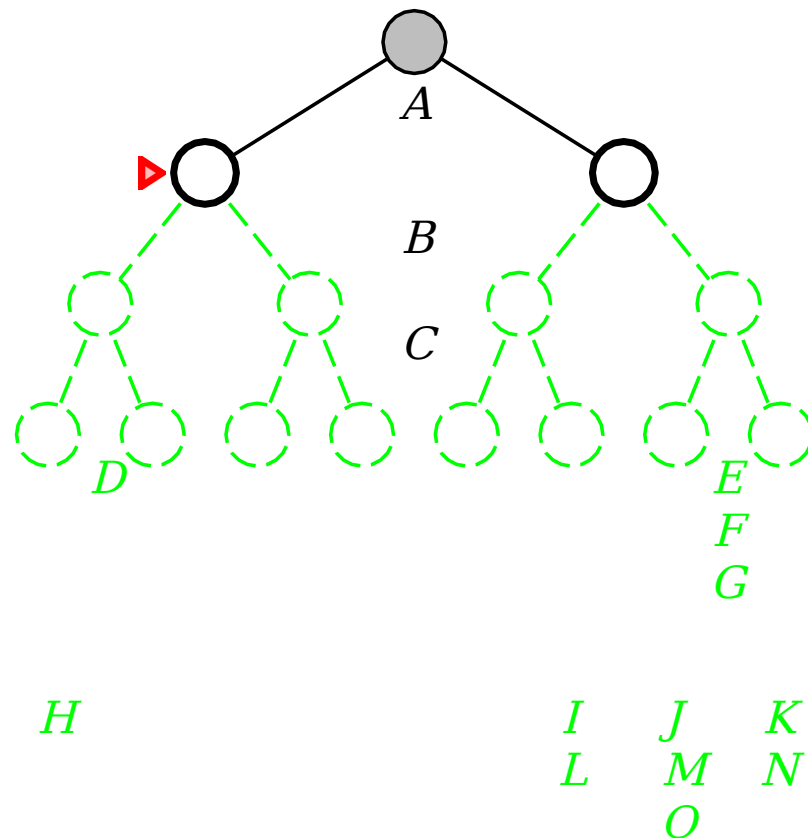


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors
at front

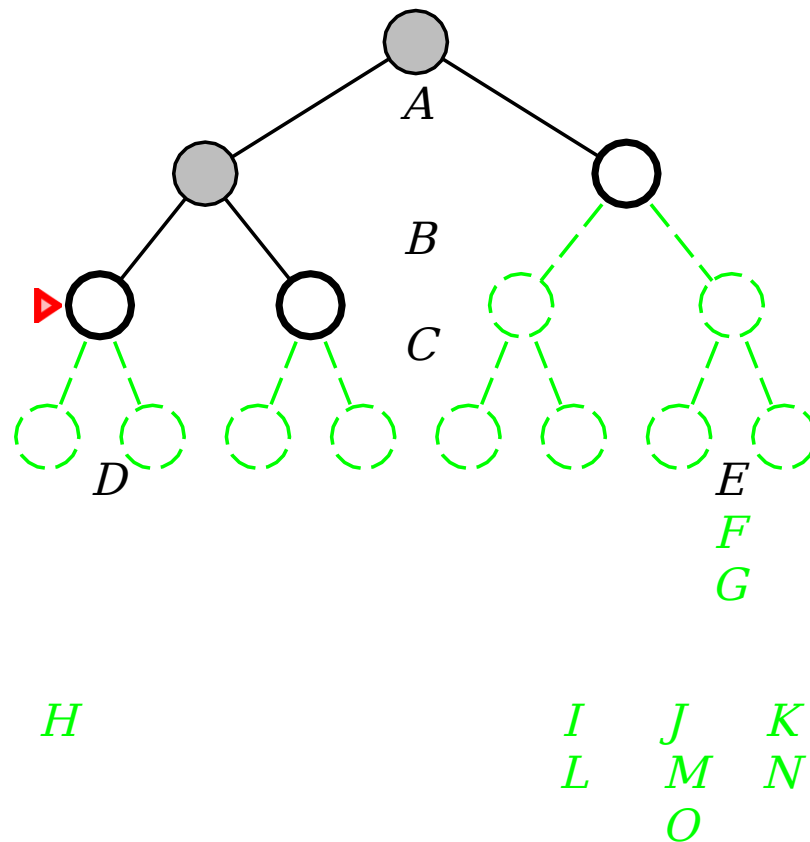


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

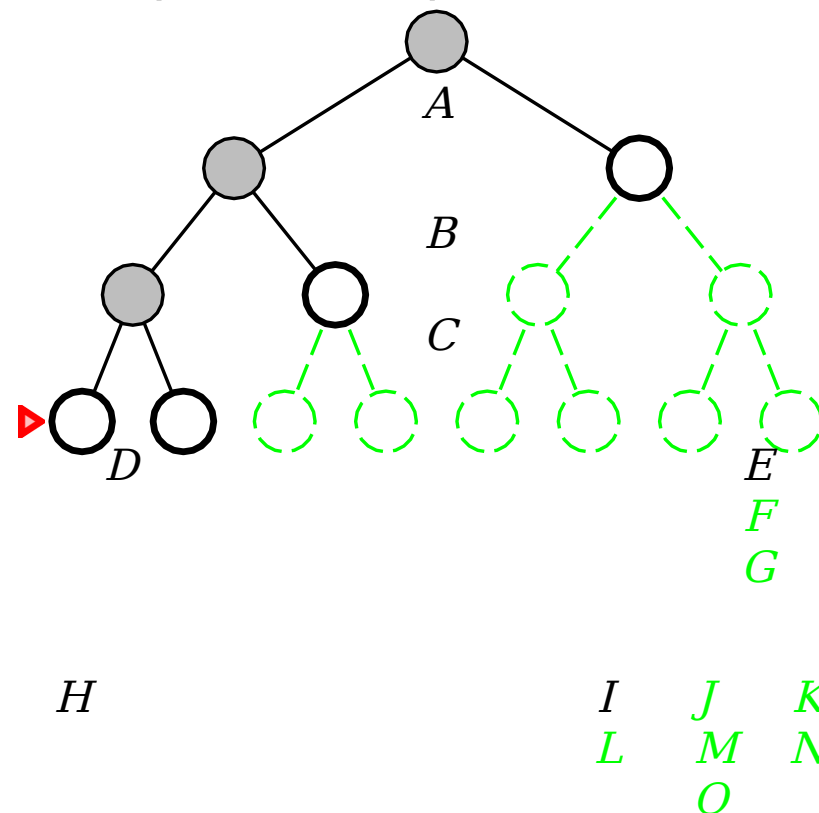


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front



Depth-first search

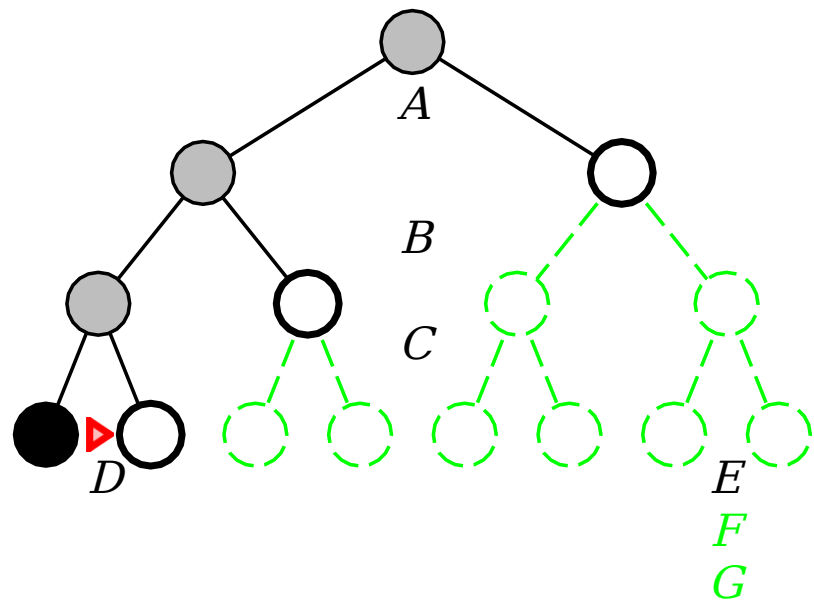
Expand deepest unexpanded node

Implementation:

Expand deepest unexpanded node

~~Implementation:~~

fringe = LIFO queue, i.e., put successors at front


$$I$$
$$J$$

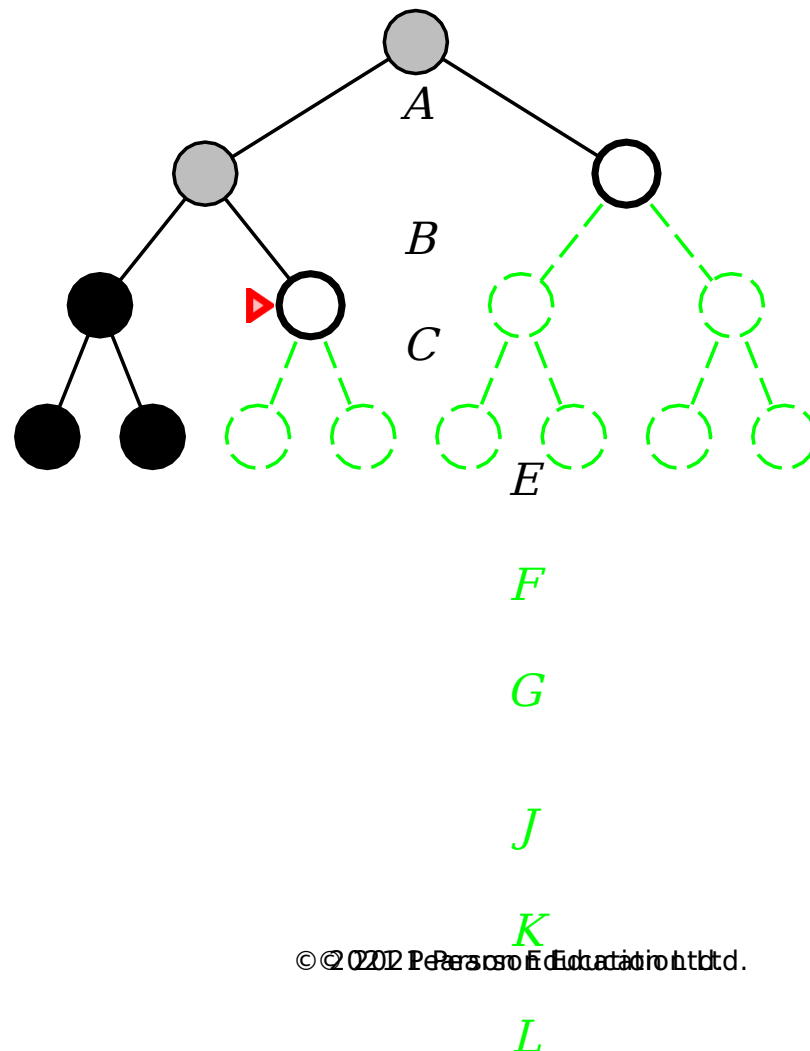
K
L
M
N
O

Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors
at front

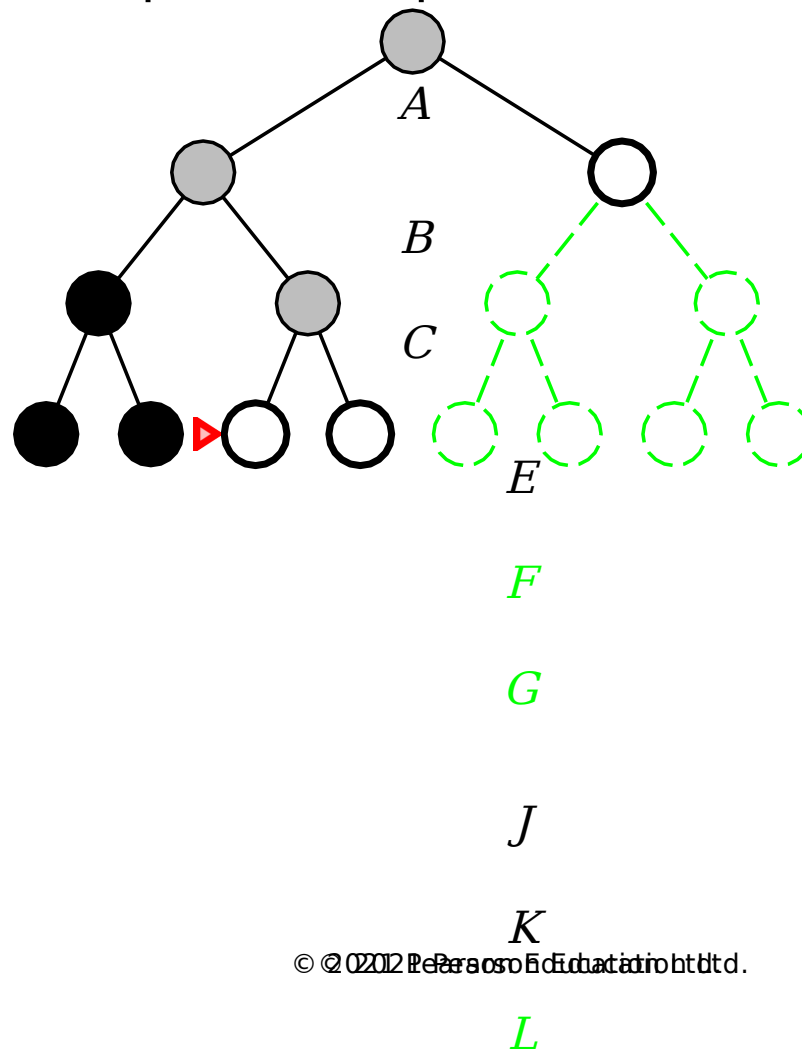


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors
at front

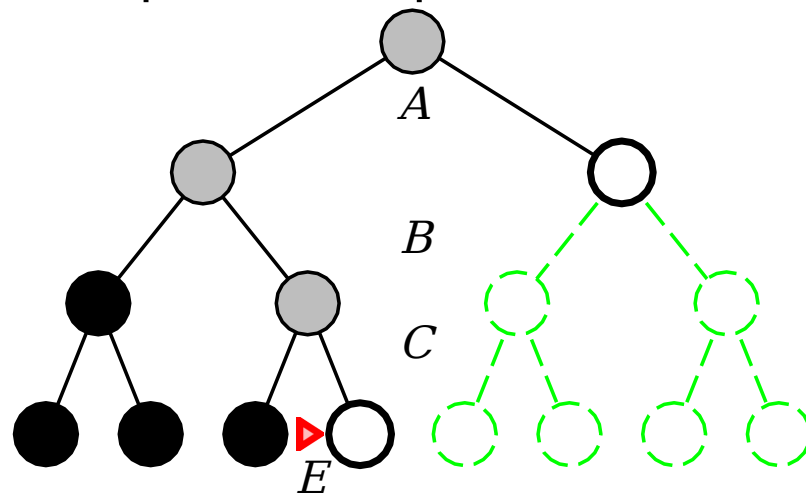


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors
at front



F

G

K

L

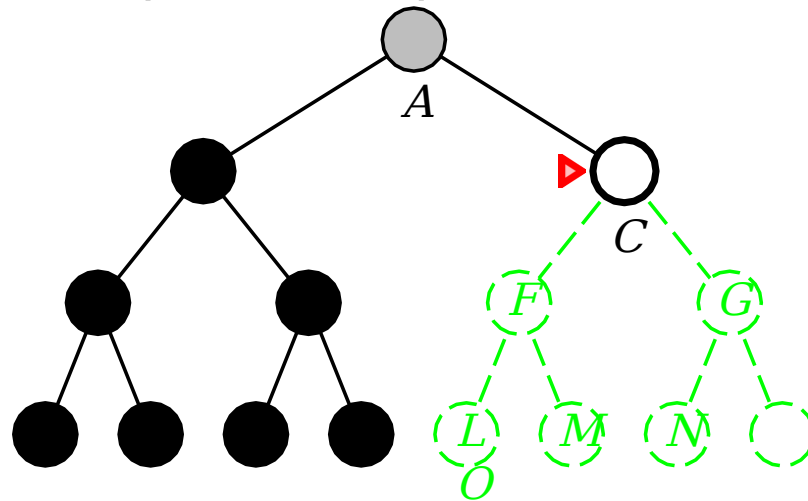
M

Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors
at front

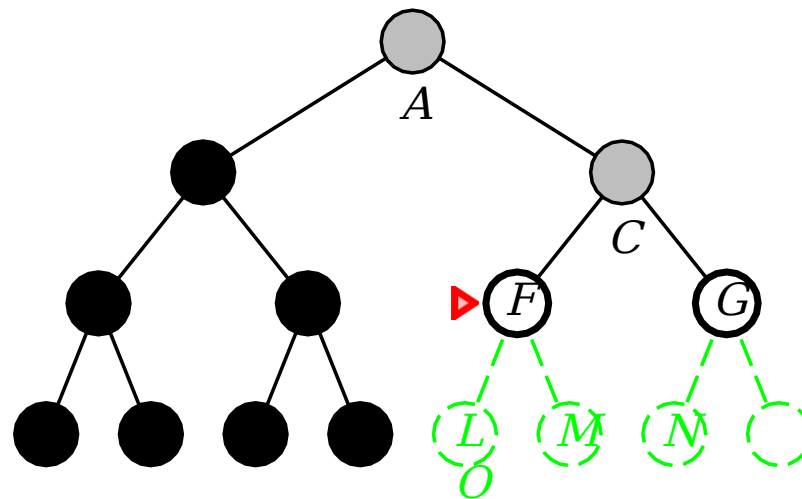


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors
at front

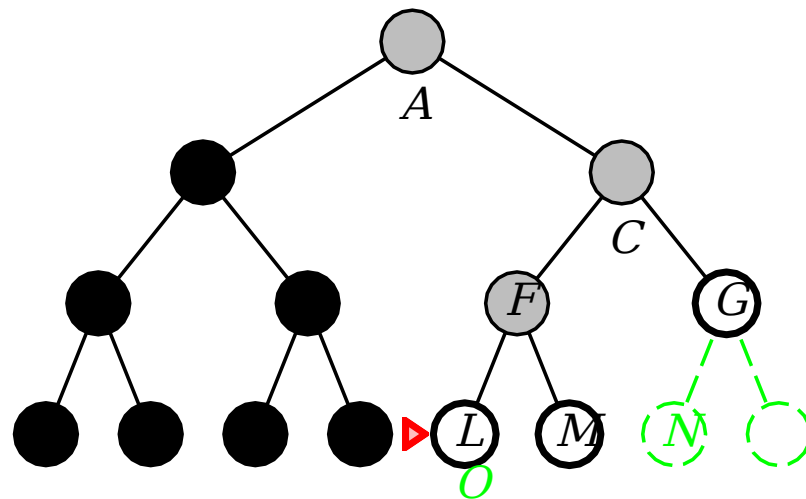


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

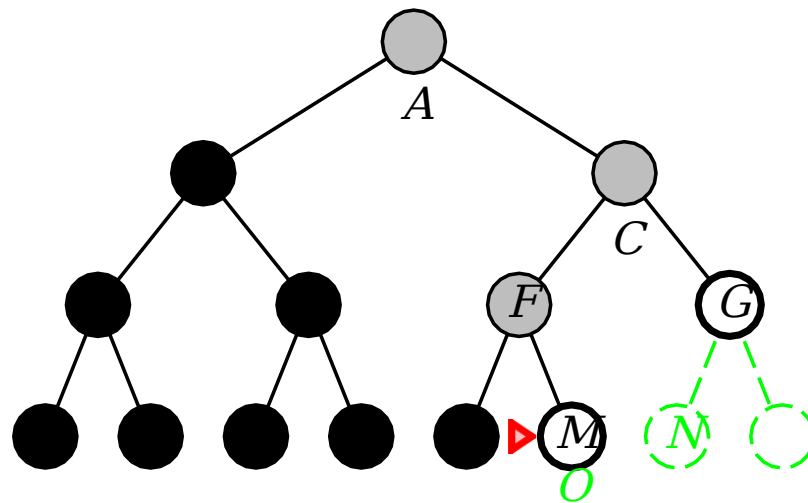


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front



Properties of depth-first search

Complete

??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
⇒ complete in finite spaces

Time??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops Modify to avoid repeated states along path
⇒ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than breadth-first

Space??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops Modify to avoid repeated states along path
⇒ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than breadth-first

Space?? $O(bm)$, i.e., linear
space! Optimal??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops Modify to avoid repeated states along path
⇒ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than breadth-first

Space?? $O(bm)$, i.e., linear

space! Optimal?? No

Depth-limited search

= depth-first search with depth limit *l*, i.e., nodes at depth *l* have no successors

Recursive implementation:

```
function Depth-Limited-Search(problem, limit) returns  
    soln/fail/cutoff
```

```
    Recursive-DLS(Make-Node(Initial-State[problem]), problem,  
        limit)
```

```
function Recursive-DLS(node, problem, limit) returns  
    soln/fail/cutoff
```

```
    cutoff-occurred? ← false
```

```
    if Goal-Test(problem, State[node]) then return node
```

```
    else if Depth[node] = limit then return cutoff
```

```
    else for each successor in Expand(node,  
        problem) do result ← Recursive-
```

```
        DLS(successor, problem, limit) if result
```

```
        = cutoff then cutoff-occurred? ← true
```

```
    if result ≠ failure then return result
```

```
    if cutoff-occurred? then return cutoff else return  
        failure
```

Iterative deepening search

function **Iterative-Deepening-Search**(*problem*) returns a solution

inputs: *problem*, a problem

for *depth* $\leftarrow$ 0 to ∞ do

result $\leftarrow$ Depth-Limited-Search(*problem*, *depth*)

 if *result* $\neq$ cutoff then return *result*

end

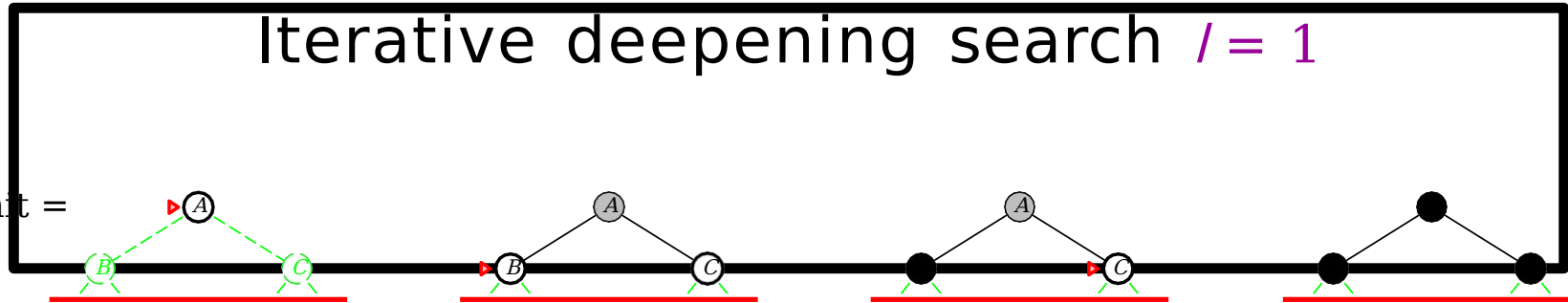
Iterative deepening search $\neq 0$

Limit =
0

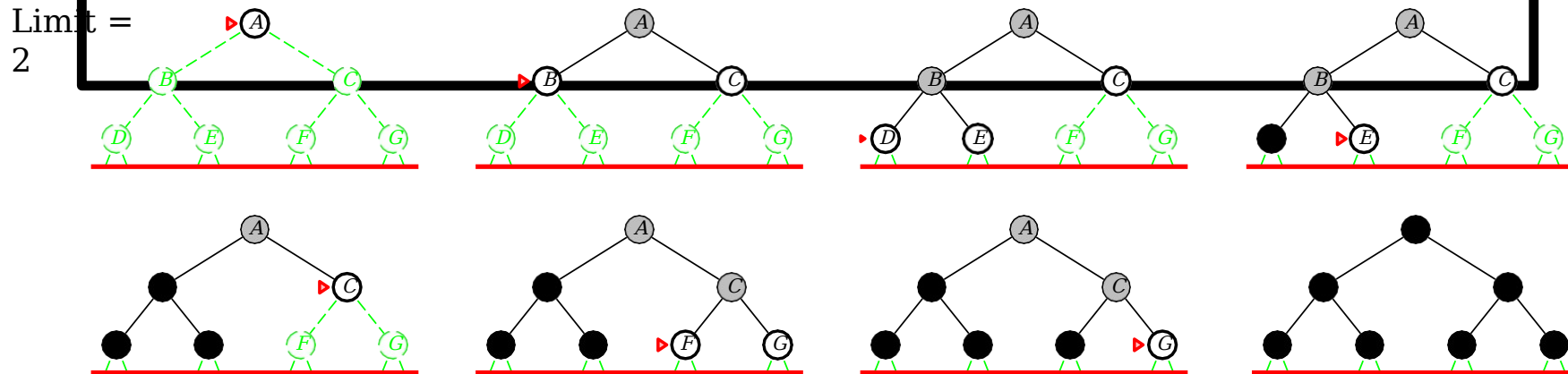


Iterative deepening search / = 1

Limit =
1

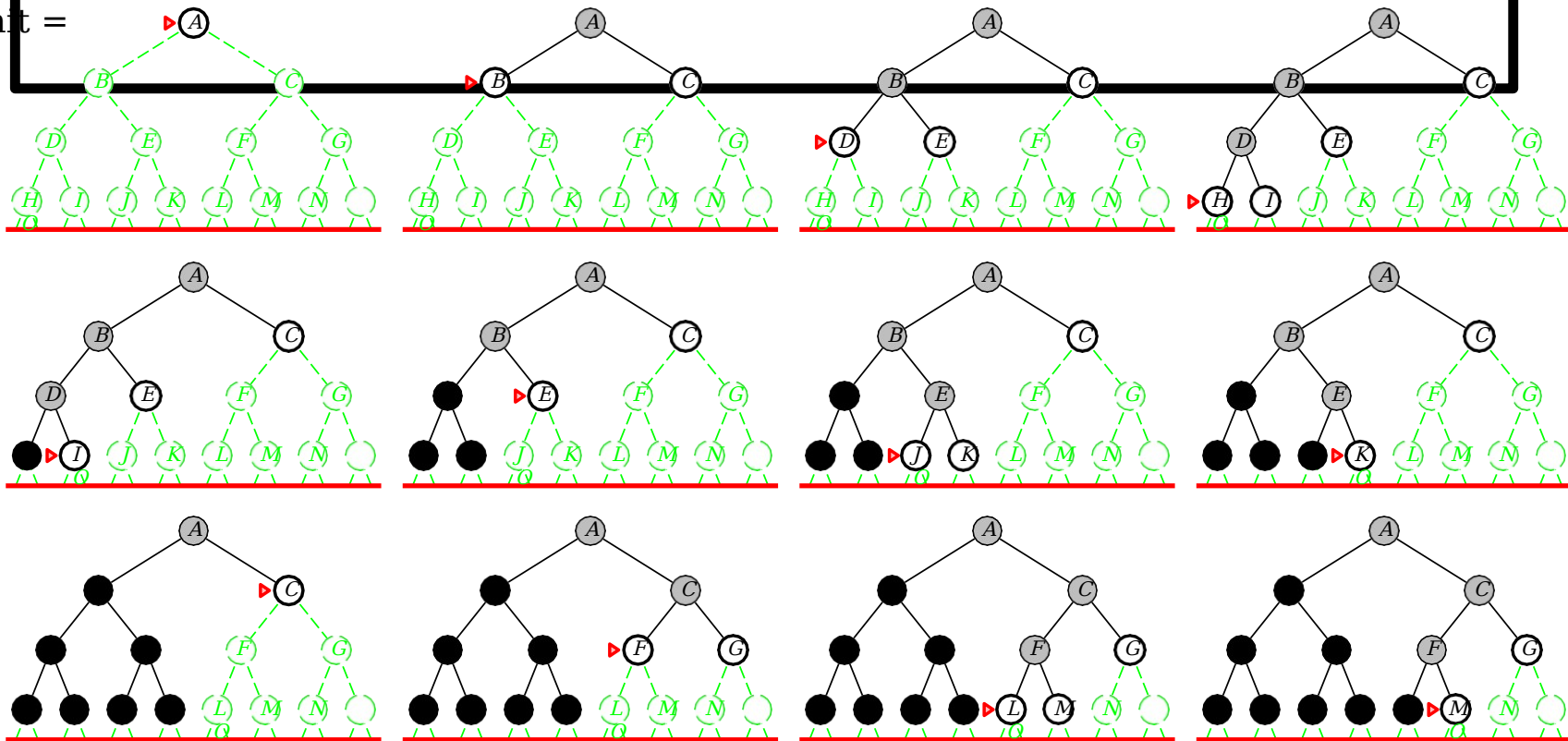


Iterative deepening search $l = 2$



Iterative deepening search / = 3

Limit =
3



Properties of iterative deepening search

Complete

??

Properties of iterative deepening search

Complete??

Yes

Time??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$

Space??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost = 1

Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

IDS does better because other nodes at depth d are not

expanded BFS can be modified to apply goal test when

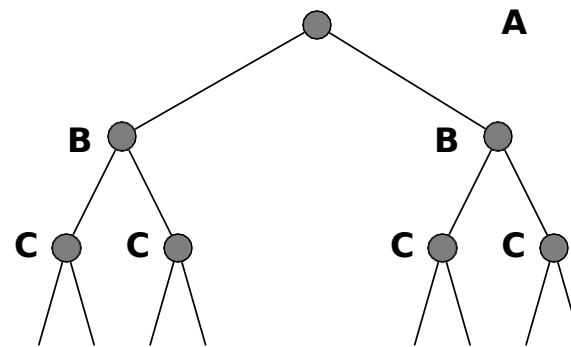
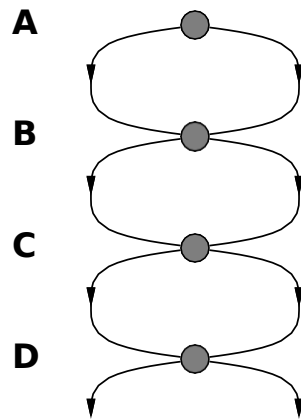
Summary of Uninformed Search algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes ¹	Yes ^{1,2}	No	No	Yes ¹	Yes ^{1,4}
Optimal cost?	Yes ³	Yes	No	No	Yes ³	Yes ^{3,4}
Time	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: ¹ complete if b is finite, and the state space either has a solution or is finite. ² complete if all action costs are $\geq \epsilon > 0$; ³ cost-optimal if action costs are all identical; ⁴ if both directions are breadth-first or uniform-cost.

Repeated states

Failure to detect repeated states can turn a linear problem into an exponential one!



Graph search

function **Graph-Search**(*problem*, *fringe*) returns a solution, or failure

```
closed ← an empty set
fringe ← Insert(Make-Node(Initial-State[problem]), fringe)
loop do
  if fringe is empty then return failure
  node ← Remove-Front(fringe)
  if Goal-Test(problem, State[node]) then return node
  if State[node] is not in closed then
    add State[node] to closed
    fringe ← InsertAll(Expand(node, problem), fringe)
end
```

Review: Tree search

```
function Tree-Search( problem, fringe) returns a  
  solution, or failure fringe ← Insert(Make-Node(Initial-  
    State[problem]), fringe) loop do  
  if fringe is empty then return failure  
  node ← Remove-Front(fringe)  
  if Goal-Test[problem] applied to State(node) succeeds  
    return node fringe ← InsertAll(Expand(node, problem),  
      fringe)
```

A strategy is defined by picking the **order of node expansion**

Best-first search

idea: use an evaluation function for each node
– estimate of “desirability”

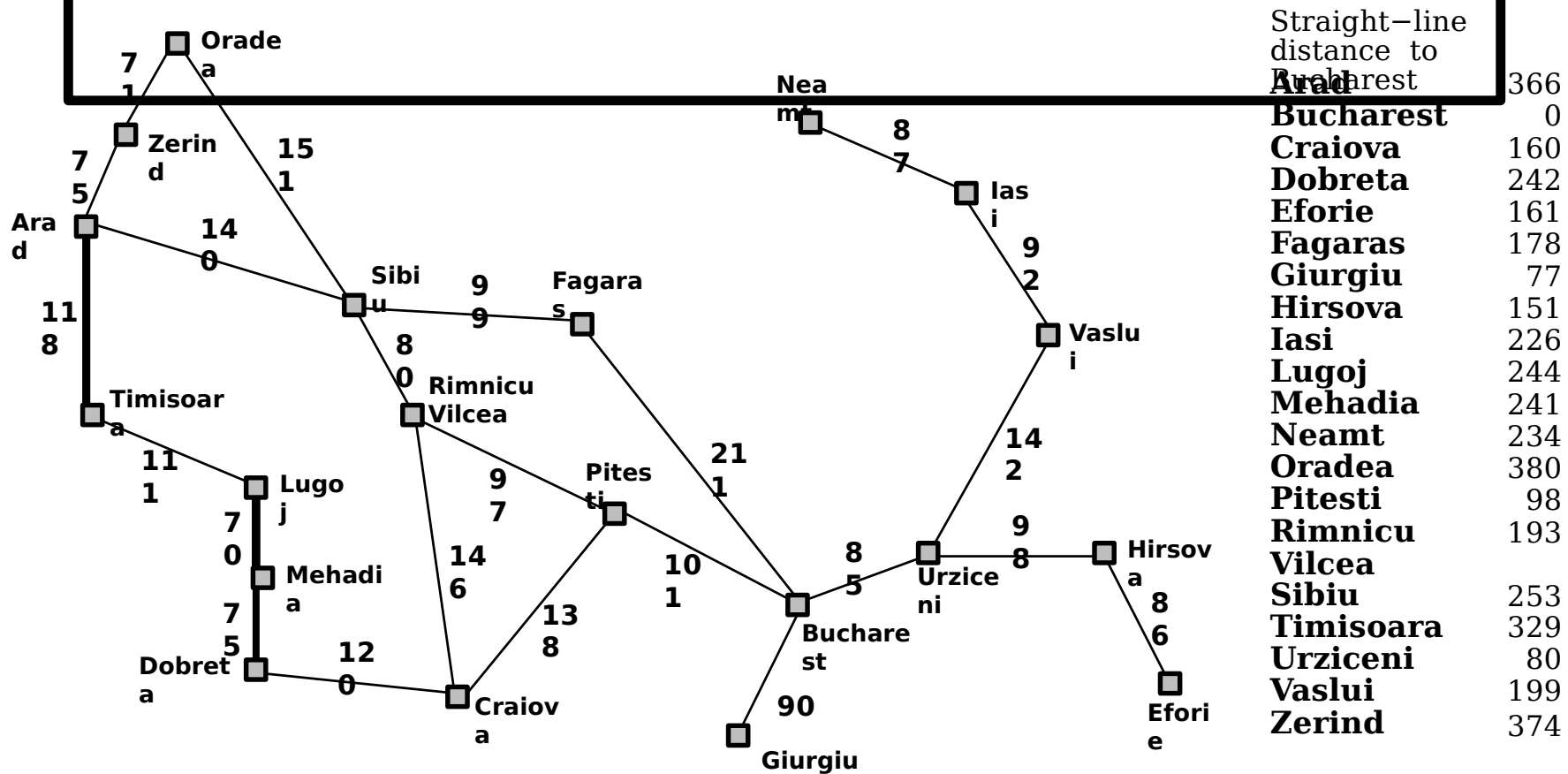
⇒ Expand most desirable

unexpanded node **Implementation:**
fringe is a queue sorted in
decreasing order of desirability

Special cases:

greedy
search A^*
search

Romania with step costs in km



Greedy search

Evaluation function $h(n)$ (heuristic)
= estimate of cost from n to the closest goal

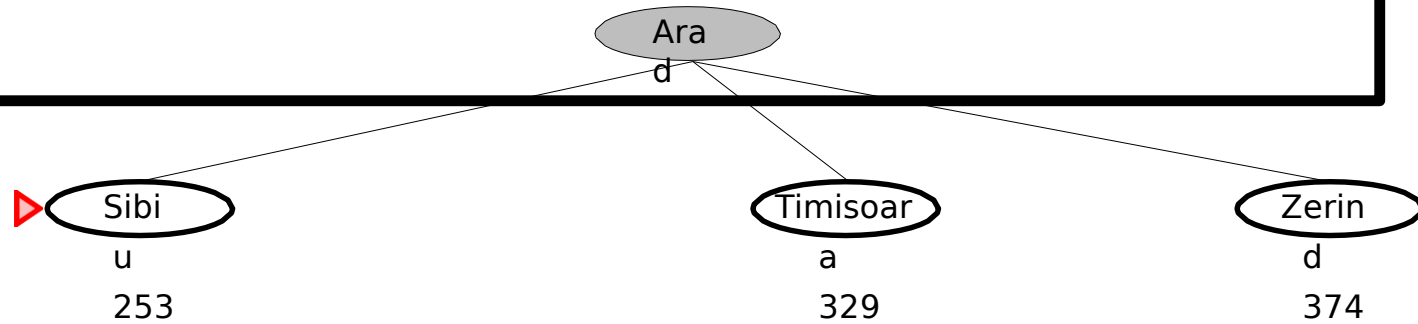
E.g., $h_{\text{SLD}}(n)$ = straight-line distance from n to Bucharest

Greedy search expands the node that appears to be closest to goal

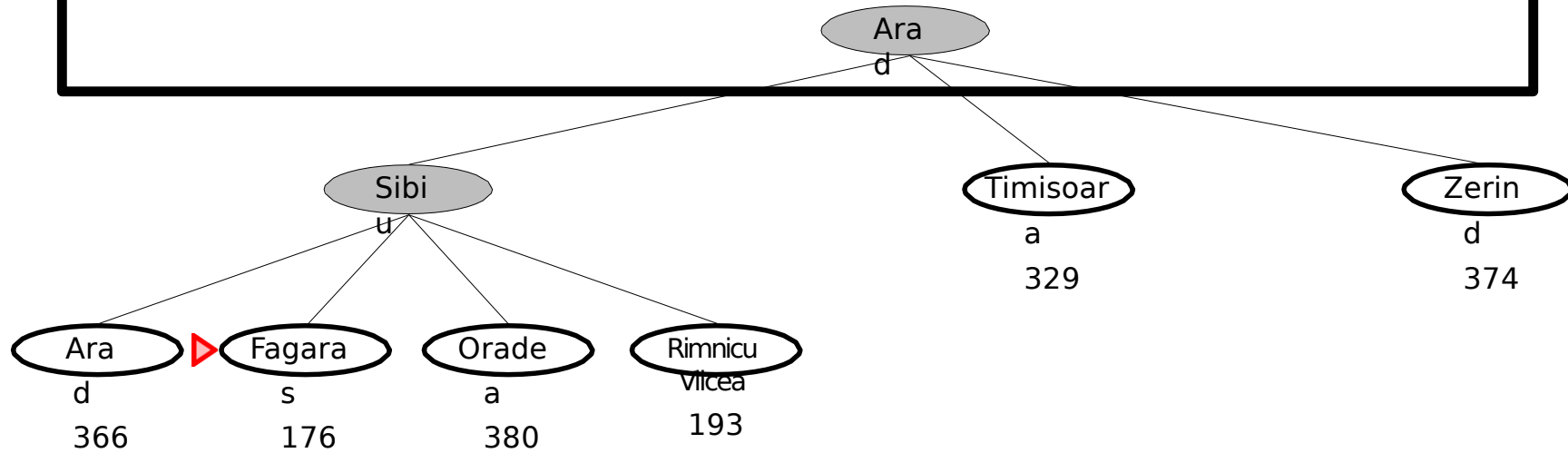
Greedy search example

▶ Ara
d
366

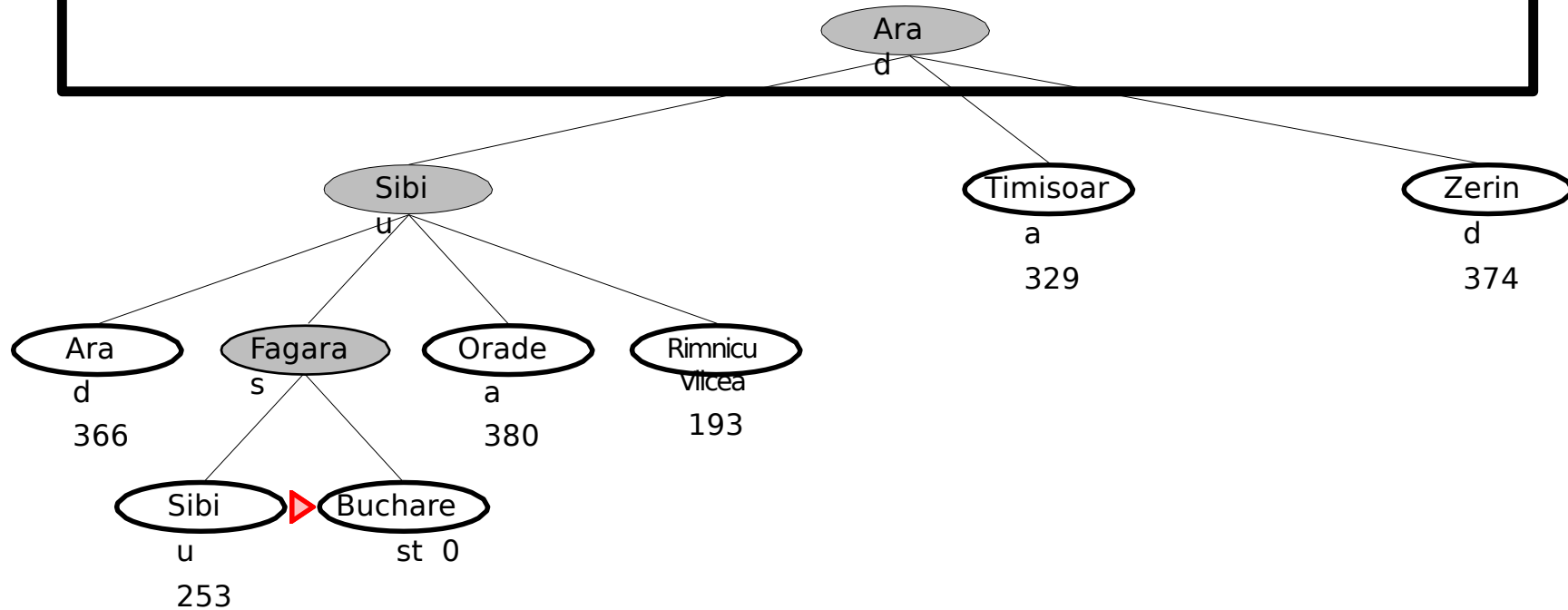
Greedy search example



Greedy search example



Greedy search example



Properties of greedy search

Complete

??

Properties of greedy search

Complete?? No—can get stuck in loops, e.g., with Oradea as goal, Iasi → Neamt → Iasi → Neamt →

Complete in finite space with repeated-state checking

Time??

Properties of greedy search

Complete?? No—can get stuck in
loops, e.g., lasi → Neamt → lasi
→ Neamt →

Complete in finite space with
repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic
improvement Space??

Properties of greedy search

Complete?? No—can get stuck in loops, e.g., lasi → Neamt → lasi → Neamt →

Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement Space?? $O(b^m)$ —keeps all nodes in memory

Optimal??

Properties of greedy search

Complete?? No—can get stuck in loops, e.g., lasi → Neamt → lasi → Neamt →

Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement Space?? $O(b^m)$ —keeps all nodes in memory

Optimal?? No

A* search

Idea: avoid expanding paths that are already expensive

Evaluation function $f(n) = g(n)$

+ $h(n)$ $g(n)$ = cost so far to

reach n

$h(n)$ = estimated cost to goal
from n

$f(n)$ = estimated total cost of
path through n to goal

A* search uses an **admissible**
heuristic

i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the **true** cost
from n . (Also require $h(n) \geq 0$, so $h(G) = 0$
for any goal G .)

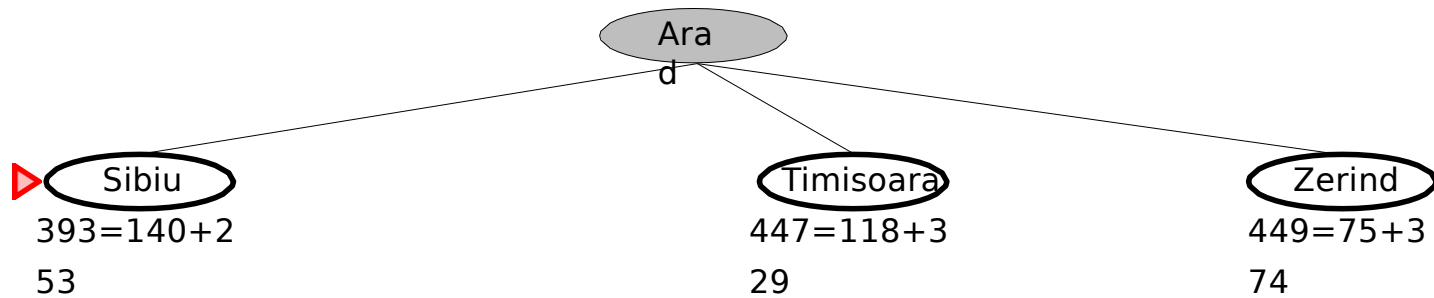
E.g., $h_{\text{SLD}}(n)$ never overestimates the actual

road distance **Theorem:** A* search is optimal

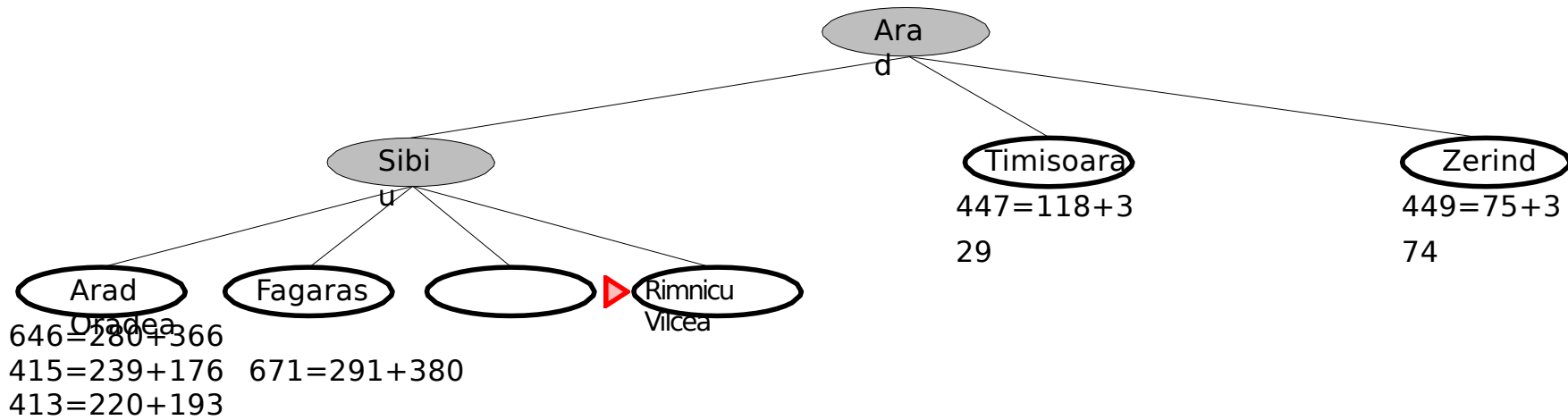
A* search example

▶ Arad
366=0+3
66

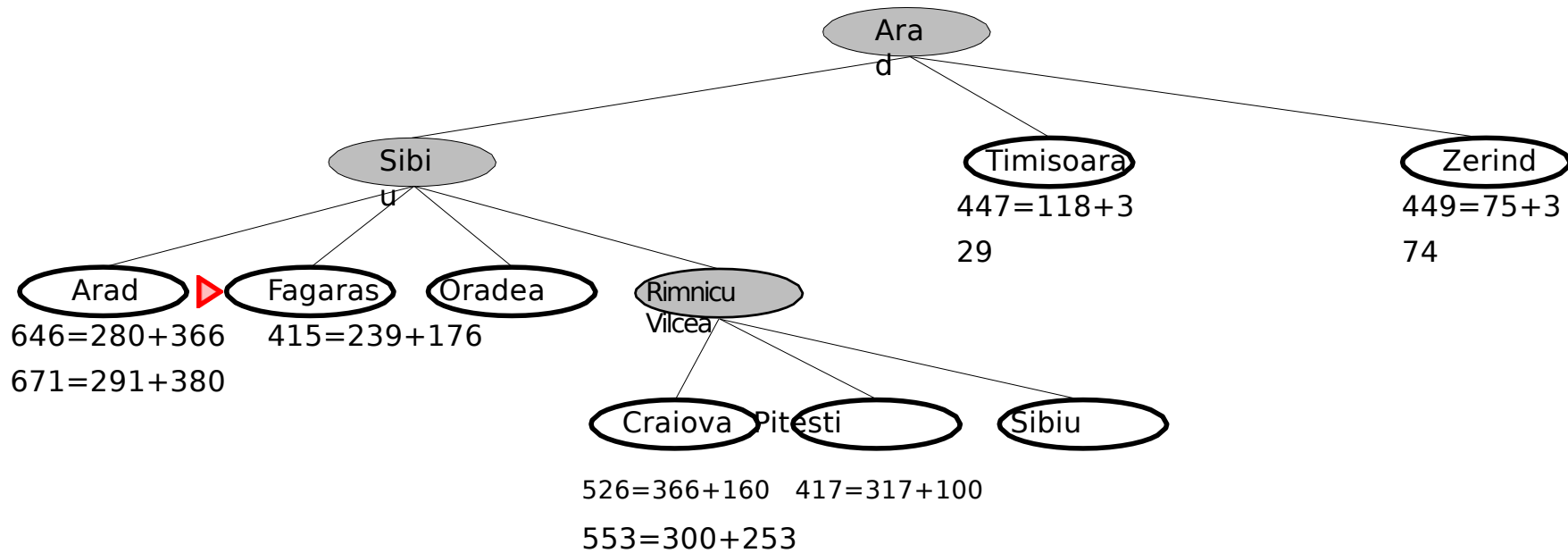
A* search example



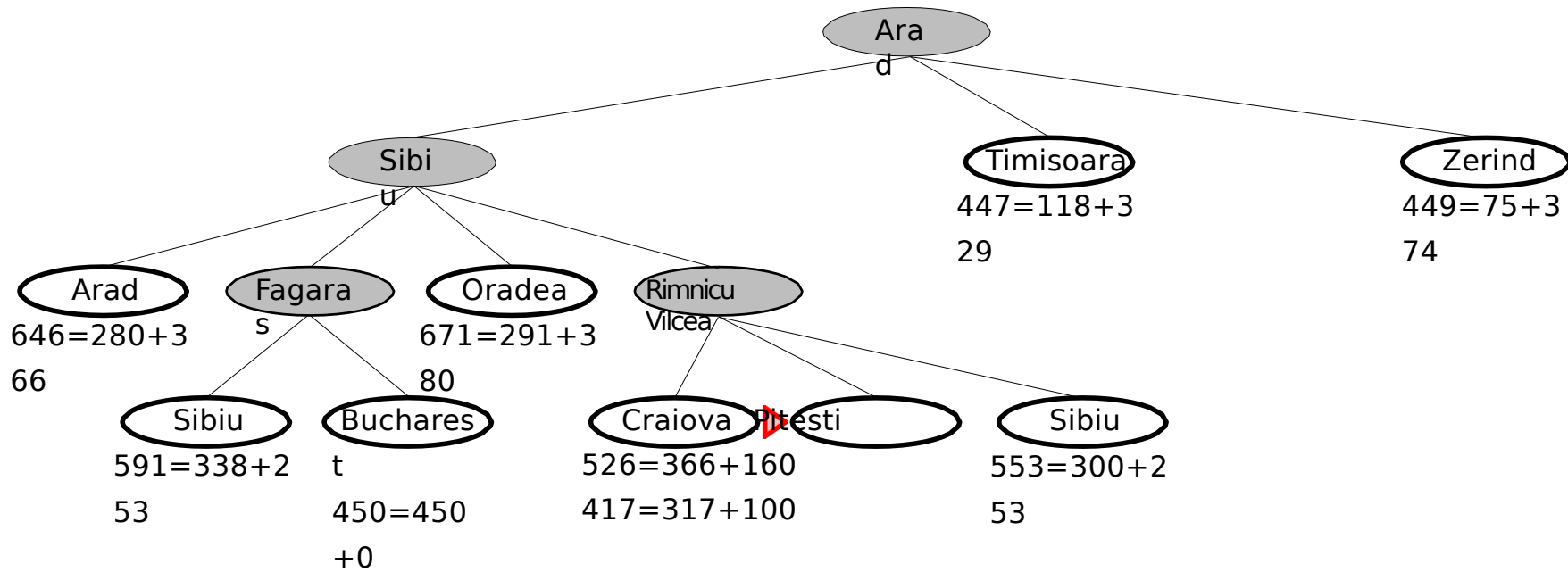
A* search example



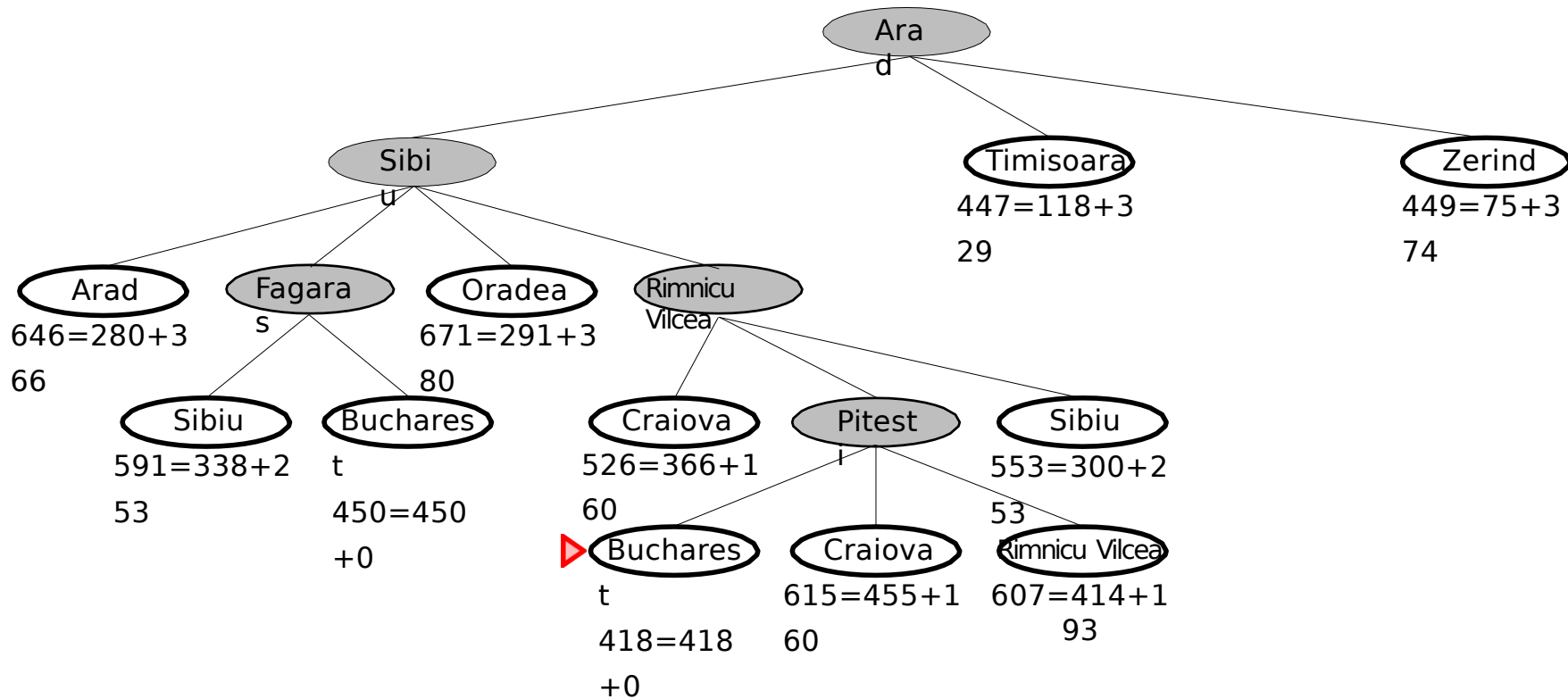
A* search example



A* search example

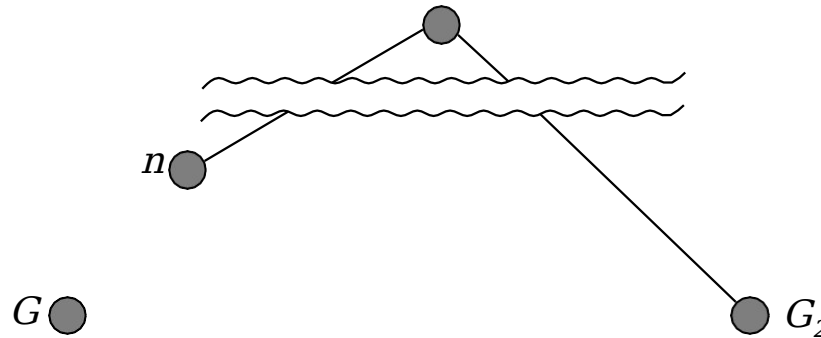


A* search example



Optimality of A* (standard proof)

Suppose some suboptimal goal G_2 has been generated and is in the queue. Let n be an unexpanded node on a shortest path to an optimal goal G_1 .



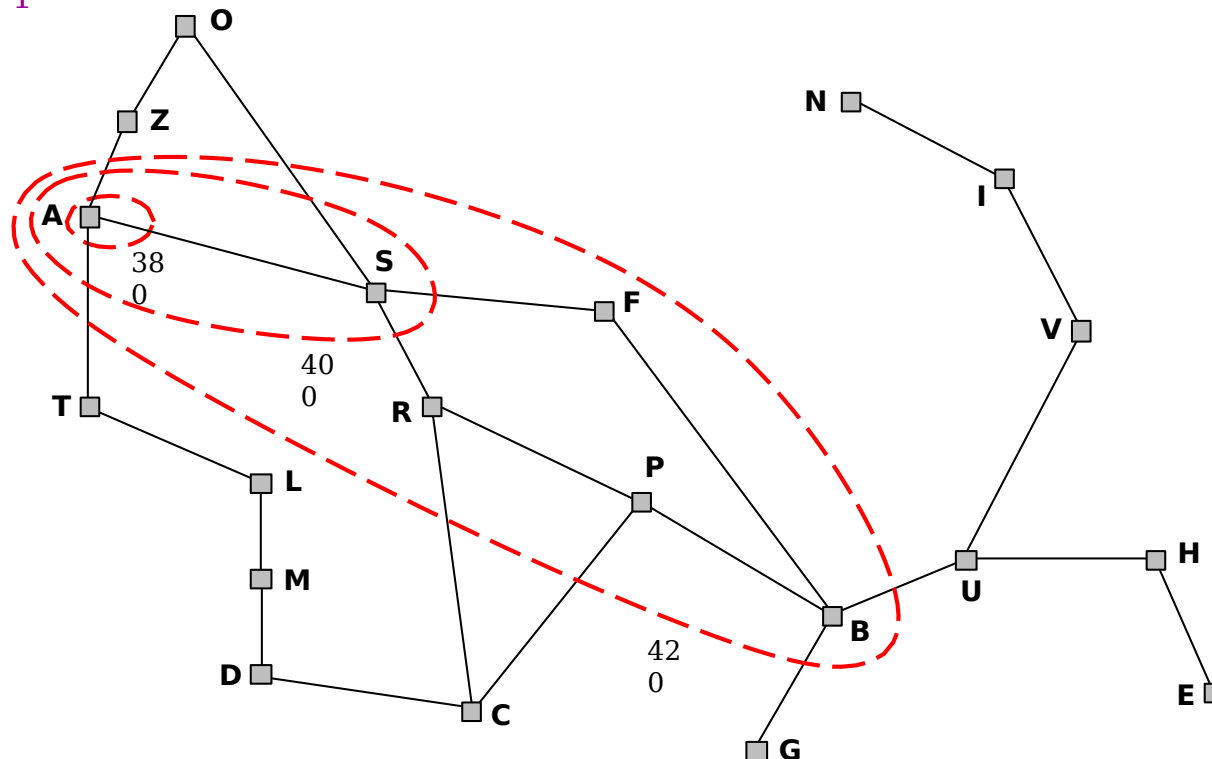
$$\begin{aligned}
 f(G_2) &= g(G_2) \quad \text{since } h(G_2) = 0 \\
 &> g(G_1) \quad \text{since } G_2 \text{ is suboptimal} \\
 &\geq f(n) \quad \text{since } h \text{ is admissible}
 \end{aligned}$$

Since $f(G_2) > f(n)$, A* will never select G_2 for expansion

Optimality of A* (more useful)

Lemma: A* expands nodes in order of increasing f value*

Gradually adds " f -contours" of nodes (cf. breadth-first adds layers) Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



Properties of A^*

Complete

??

Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f^*(G)$

Time??

Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in $h \times$ length of soln.] Space??

Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in $h \times$ length of soln.] Space?? Keeps all nodes in memory

Optimal??

Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in $h \times$ length of soln.] Space?? Keeps all nodes in memory

Optimal?? Yes—cannot expand f_{i+1} until f_i is finished A* expands all nodes with $f(n) <$

C^*

A* expands some nodes with $f(n) = C^*$

A* expands no nodes with $f(n) > C^*$

Proof of lemma: Consistency

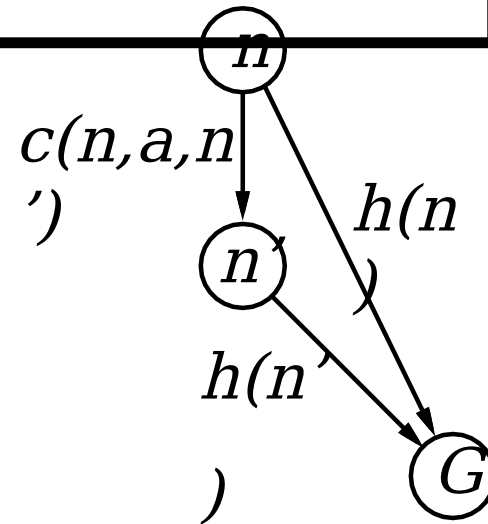
A heuristic is **consistent** if

$$h(n) \leq c(n, a, n') + h(n')$$

If h is consistent, we have

$$\begin{aligned} f(n) &= g(n) + h(n) \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n') + h(n') \\ &= f(n') \end{aligned}$$

I.e., $f(n)$ is nondecreasing along any path.



Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)

7	2	4
5		6
8	3	1

Start
State

1	2	3
4	5	6
7	8	

Goal
State

$$\frac{h_1(S)}{=??}$$

$$\frac{h_2(S)}{=??}$$

Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total **Manhattan** distance

(i.e., no. of squares from desired location of each tile)

7	2	4
5		6
8	3	1

Start
State

1	2	3
4	5	6
7	8	

Goal
State

$$h_1(S) = ?? \quad 6$$

$$h_2(S) = ?? \quad 4+0+3+3+1+0+2+1$$

$$= 14$$

Dominance

If $h_2(n) \geq h_1(n)$ for all n (both admissible), then h_2 dominates h_1 and is better for search

Typical search costs:

$d = 14$ IDS = 3,473,941 nodes

$A^*(h_1) = 539$

nodes $A^*(h_2) =$

113 nodes

$d = 24$ IDS $\approx$ 54,000,000,000

nodes

$A^*(h_1) = 39,135$

nodes $A^*(h_2) =$

1,641 nodes

Given any admissible heuristics

$$h_a, h_b, h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates

Relaxed problems

Admissible heuristics can be derived from the **exact** solution cost of a **relaxed** version of the problem

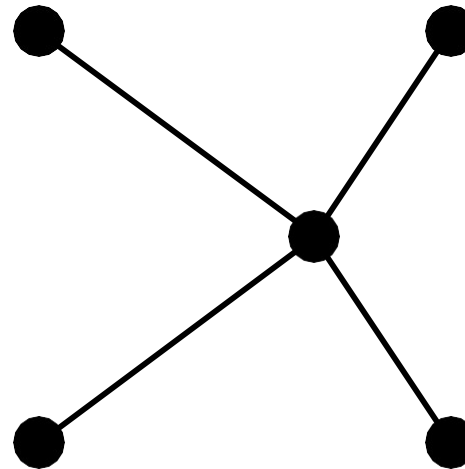
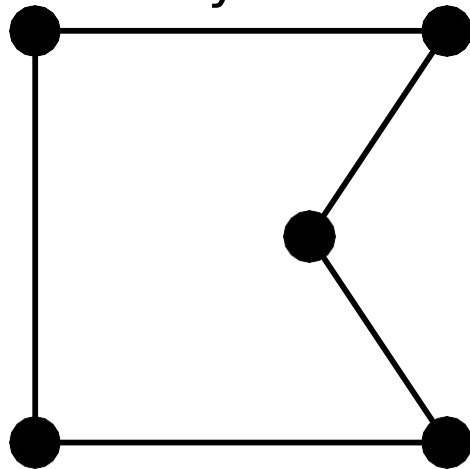
If the rules of the 8-puzzle are relaxed so that a tile can move **anywhere**, then $h_1(n)$ gives the shortest solution

If the rules are relaxed so that a tile can move to **any adjacent square**, then $h_2(n)$ gives the shortest solution

Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

Relaxed problems contd.

Well-known example: **travelling salesperson problem** (TSP) Find the shortest tour visiting all cities exactly once



Minimum spanning tree can be computed in $O(n^2)$ and is a lower bound on the shortest (open) tour

Summary

A problem consists of five parts: the **initial state**, a set of **actions**, a **transition model** describing the results of those actions, a set of **goal states**, and an **action cost function**.

Uninformed search methods have access only to the **problem definition**. Algorithms build a search tree in an attempt to find a solution.

Informed search methods have access to a **heuristic** function $h(n)$ that estimates the cost of a solution from n .